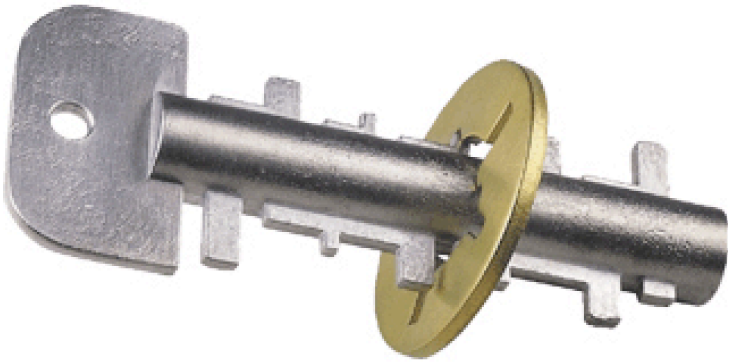
The Mathematica® Journal

Maze for Free the Key Puzzle

Kenneth E. Caviness



▲ **Figure 1.** "Free the Key" puzzle [1]

The author has written a series of guided tours showing how to visualize and solve puzzles programmatically, creating animated visualizations, showcasing various programming tricks and algorithms, and using some good old-fashioned physics problem-solving strategies with an occasional foray into abstract mathematics. Here the "Free the Key" puzzle is solved and animated together with basically equivalent (read *isomorphic*) alternative representations.

Introduction

Description of the Puzzle

"Free the Key" is a mechanical puzzle designed by Oskar van Deventer [2]. The object is to get the disk off the key, or to put it another way, to pull the key out of the disk. But some of the teeth are too tall to go through some of the slits in the disk, so you may have to turn the key (or the disk). Of course, sometimes the disk cannot turn and you must slide it to another position first!

Comments

The surprising thing about this puzzle is that sometimes it is impossible to move the disk closer to the end without backing up first, moving it farther away from the end. But it is not a particularly hard puzzle once one accepts the need to back up at times. Also, it is exactly as hard to put the disk back on the key as it is to take it off.

After solving the puzzle a few times, it began to feel like a maze to me. At any position you have four options: you can slide the disk left or right, or turn it counterclockwise or clockwise. That is analogous to moving left, right, up, or down in a maze. Obviously in a two-dimensional maze the path out may wander a lot, so we should not be surprised if the disk has to move left at some positions before we find a path leading farther to the right.

I decided to write a *Mathematica* program to visualize the puzzle, and at the same time investigate the possible relationship between key-disk puzzles and mazes.

The Program

Create Key and Disk Arrays

In order to create a virtual version of the puzzle we need some way to talk about the height of the teeth and compare them to the height of the cuts in the disk. I took height 0 to mean the tooth does not emerge from the cylinder at all, 1 for the shortest height tooth above the level of the cylinder, then 2, 3, and 4 for successively higher teeth. The variable `key` is made of two lists, the heights of the teeth above and below the cylinder. The list `disk` has the heights of the cuts in the disk, defining as height *n* the height of the shortest slit that a tooth of height *n* can pass through. Again, the heights range from 0 to 4.

```
key = {{0, 1, 1, 3, 0, 1, 1, 2, 0, 1, 1, 2, 0, 1, 1, 3, 0,
    1, 1, 4, 0}, {0, 4, 0, 1, 1, 2, 1, 1, 1, 3, 0, 1, 1,
    2, 1, 1, 1, 3, 0, 1, 0}};

disk = {
   4, 0, 1, 3, 1, 1, 4, 0,
   1, 1, 2, 1, 0, 2, 1, 1
  };
```

Notice that `disk` is a single list, but we need to look at two parts of it at once, to see whether both the tooth above and the tooth below can fit. For human convenience I have divided it up on two lines, but for the purposes of the program it is a single list.

□ Create Visual Image of Puzzle

▪ *Routines to Facilitate Construction of 3D Objects*

Here is a function to convert `Disk` and `Rectangle` objects into `Polygon` objects and to handle lists of these objects. The function `Polygonize` ignores all other objects, simply leaving them unchanged.

```
Polygonize[Disk[{x_, y_}, r_]] :=
  Polygon[Table[{x, y} + r {Cos[t], Sin[t]},
    {t, 0.0, 2 Pi, 2 Pi / 20}]];

Polygonize[Rectangle[{xmin_, ymin_}, {xmax_, ymax_}]] :=
  Polygon[{{xmin, ymin}, {xmax, ymin}, {xmax, ymax},
    {xmin, ymax}}];

Polygonize[l_List] := Polygonize /@ l;

Polygonize[x___] := x;
```

The function `Cylindrize` acts on a 2D `Polygon` (with coordinates as lists of length two) and a vector indicating the direction to extrude it, drawing it into a third dimension. The default is to turn the 2D object into a 3D object with height 1 in the z direction.

```
Cylindrize[Polygon[p_List], v_List: {0, 0, 1}] :=
 Cylindrize[Polygon[p /. {x_, y_} →
      If[v[[1]] == 0 && v[[2]] == 0, {x, y, 0},
       If[v[[1]] == 0 && v[[3]] == 0, {x, 0, y},
        If[v[[2]] == 0 && v[[3]] == 0, {0, x, y},
         {x, y, 0}]]]
   ], v] /; (d = Dimensions[p];
   d[[1]] ∈ Integers && d[[2]] == 2 && Dimensions[v] == {3})

Cylindrize[Polygon[p_List], v_List: {0, 0, 1}] :=
 Join[
   {Polygon[p], Polygon[v + # & /@ p]},
   MapThread[
    Polygon[{#1, #2, #2 + v, #1 + v}] &,
    {p, Append[Rest[p], First[p]]}
   ]
  ] /; (d = Dimensions[p];
   d[[1]] ∈ Integers && d[[2]] == 3 && Dimensions[v] == {3})
```

```
Cylindrize[d_Disk, v___] := Cylindrize[Polygonize[d], v];
Cylindrize[r_Rectangle, v___] :=
  Cylindrize[Polygonize[r], v];
Cylindrize[l_List, v___] := Cylindrize[#, v] & /@ l;
```

■ Cylindrical Hub of the Key

We suppress the default box around `Graphics3D` objects throughout this notebook. For the hub of the key we form a cylinder of radius 2 and length at least 21 to accommodate the teeth. As with all `Graphics3D` objects, we can easily rotate and view the cylinder from all sides using the mouse.

```
SetOptions[Graphics3D, Boxed -> False];
```

```
g1=Graphics3D[Cylinder[{{0,0,-3},{0,0,22}},2]]
```

■ Key Teeth

From the key height data we construct the visual representation of the teeth. It is convenient to list together the x position with the heights needed above and below. Since the cylinder radius is 2, all heights are at least 2 and all depths at most -2. For instance, both key lists begin with 0, so the first element (which we label by $n = 0$) of the `boxinfo` list is $\{0, 2, -2\}$. We can build the whole list this way.

```
key
```

```
{{0, 1, 1, 3, 0, 1, 1, 2, 0, 1, 1, 2, 0, 1, 1, 3, 0, 1, 1, 4, 0},
 {0, 4, 0, 1, 1, 2, 1, 1, 1, 3, 0, 1, 1, 2, 1, 1, 1, 3, 0, 1, 0}}
```

```
{1, -1} (2 + key)
```

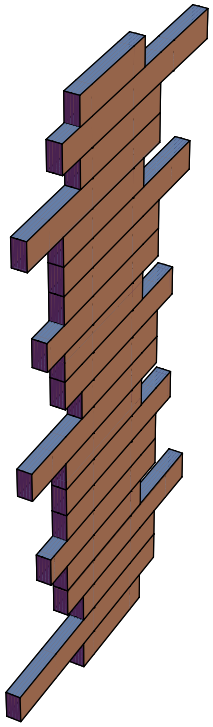
```
{{2, 3, 3, 5, 2, 3, 3, 4, 2, 3, 3, 4, 2, 3, 3, 5, 2, 3, 3, 6, 2},
 {-2, -6, -2, -3, -3, -4, -3, -3, -3, -5,
  -2, -3, -3, -4, -3, -3, -3, -5, -2, -3, -2}}
```

```
boxinfo = Transpose@Prepend[%, Range[0, 20]]
```

```
{{0, 2, -2}, {1, 3, -6}, {2, 3, -2}, {3, 5, -3},
 {4, 2, -3}, {5, 3, -4}, {6, 3, -3}, {7, 4, -3}, {8, 2, -3},
 {9, 3, -5}, {10, 3, -2}, {11, 4, -3}, {12, 2, -3},
 {13, 3, -4}, {14, 3, -3}, {15, 5, -3}, {16, 2, -3},
 {17, 3, -5}, {18, 3, -2}, {19, 6, -3}, {20, 2, -2}}
```

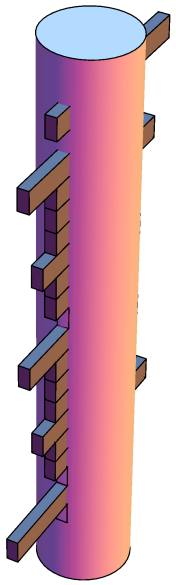
Now we build a collection of `Cuboid` objects using `boxinfo`. Here we let the x coordinates be -0.25 and 0.25, the z coordinates be given by n and $n+1$, and the y coordinates be the heights and depths, all listed in the `boxinfo` array.

```
g2 =
 Graphics3D[Join[{GrayLevel[0.5], EdgeForm[GrayLevel[0]]},
   (Cuboid[{0.25, #1[[3]] + 0.1, #1[[1]]},
       {-0.25, #1[[2]] - 0.1, #1[[1]] + 1}] &) /@ boxinfo]]
```



Mathematica has excellent hidden-line routines, letting us combine these two graphics objects.

```
Show[{g1, g2}]
```



Drag the key to rotate it. **Ctrl**+drag zooms in or out.

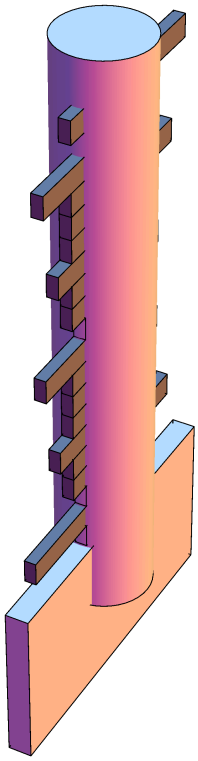
■ *Key Handle*

Let us add a handle at the near end, a `Rectangle`, cylindrized to give it thickness 1 in the *x* direction and slid back 0.5 to center it.

```
g3 = (Graphics3D@
      Cylindrize[
       Rectangle[{-7, -7}, {7, 0}],
       {1, 0, 0}] /. {x_, y_, z_} → {x - .5, y, z});
```

■ Entire Key

```
gkey = {g1, g2, g3}; Show[gkey]
```

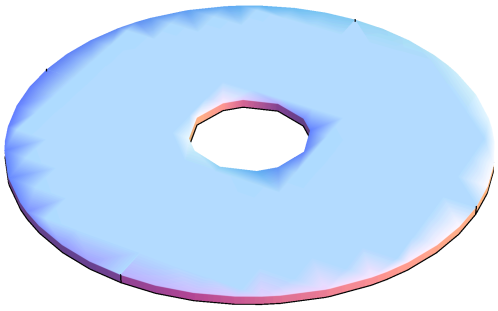


This is the model of the key.

■ Disk

The easiest way to create the disk is to use the function `RegionPlot3D`, which lets us specify equations or inequalities for the three-dimensional region we want to include in the object being graphed. For instance, the following generates a washer with thickness 0.4, inner radius 2, and outer radius 8.

```
RegionPlot3D[0 ≤ z ≤ 0.4 && 2 ≤ Sqrt[x^2 + y^2] ≤ 8,
 {x, -8, 8}, {y, -8, 8}, {z, 0, 0.5},
 Mesh → None, Axes → None, Boxed → False,
 BoxRatios → {16, 16, 0.4}]
```



We have allowed $r = \sqrt{x^2 + y^2}$ to take on all values between 2 and 8. In order to cut out the notches in the disk, we need to increase the inner radius by the appropriate value of `disk`.

```
disk
```

```
{4, 0, 1, 3, 1, 1, 4, 0, 1, 1, 2, 1, 0, 2, 1, 1}
```

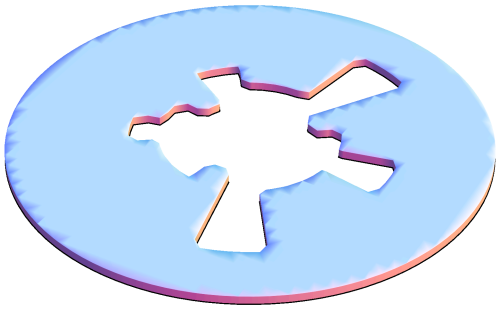
There are 16 regions to notch.

```
Length[disk]
```

```
16
```

The function `ArcTan[x, y]` tells us the angle (in radians) that corresponds to x and y. Dividing by 2π (a full circle or 360°) and multiplying by 16 scales the angle for our use, but returns values between -8 and 8 rather than between 0 and 16. We use `Round` and `Mod[…, n, 1]` to shift to values between 1 and n. We ask for this entry of the disk array and add it to 2 to give a new inner radius for each of the 16 regions. Let us also scale the r values by a factor of 1.05 to effectively expand the ring by 5% so it clearly fits on the key.

```
vdisk =
 RegionPlot3D[
  0 ≤ z ≤ 0.4 &&
   2 + disk[[Mod[Round[16 * ArcTan[y, x] / (2 π)], 16, 1]]] ≤
    Sqrt[x^2 + y^2] / 1.05 ≤ 8,
  {x, -8.4, 8.4}, {y, -8.4, 8.4}, {z, 0, 0.5},
  Mesh → None, Axes → None, Boxed → False, PlotPoints → 40,
  BoxRatios → {2 × 8.4, 2 × 8.4, 0.4}]
```

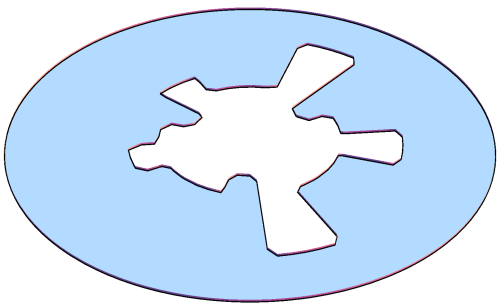


It will be more convenient later if we have a simple function to display the disk in all 16 rotational positions and all 21 translational positions. The offsets are to orient the disk properly on the first key segment, for the initial values $i = j = 1$.

```
gdisk[i_Integer, j_Integer] := gdisk[i, j] =
  vdisk /. GraphicsComplex[p_List, rest__] :> GraphicsComplex[
     TranslationTransform[{0, 0, i - 0.6}]@
      RotationTransform[2 π j / 16, {0, 0, 1}]@p, rest]
```

Here is the first orientation of our disk.

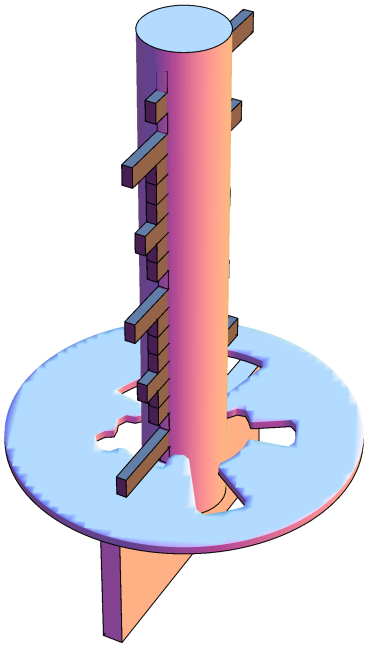
```
gdisk[1, 1]
```



Now let us display it on the key. Turn the image and look at it from the side to verify that

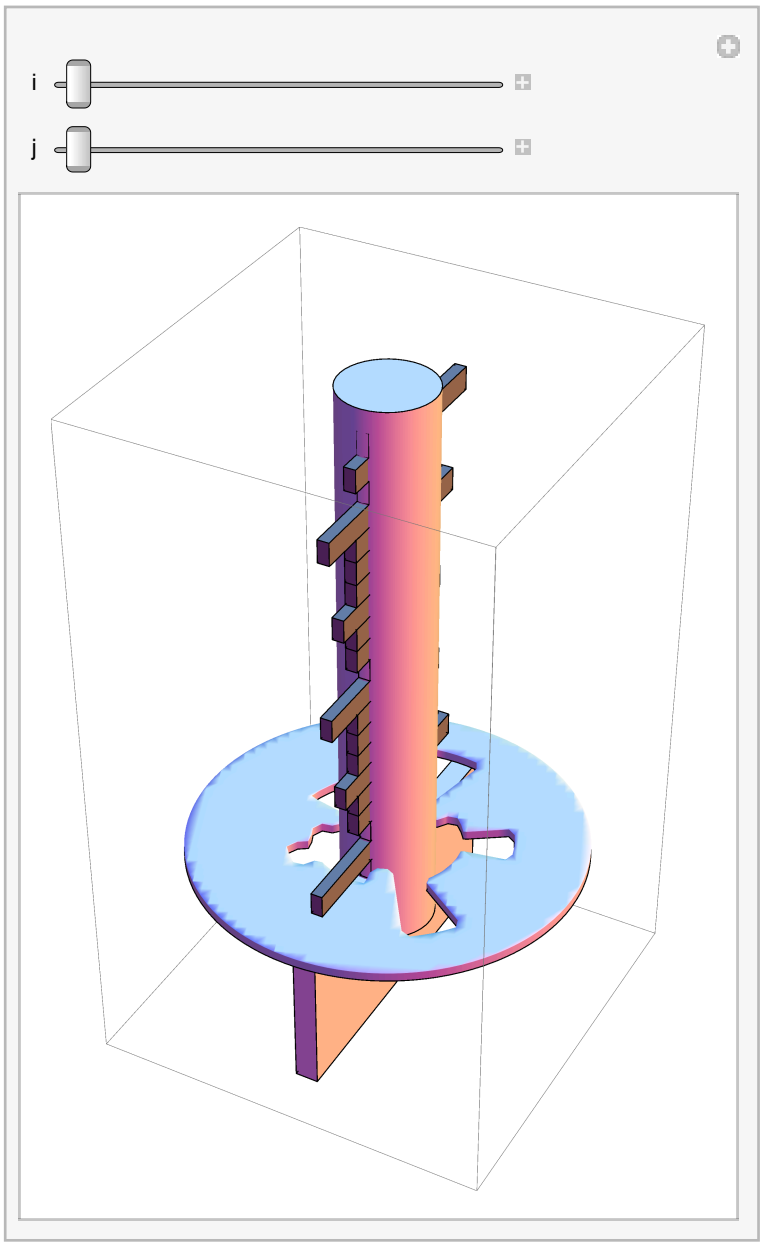
the disk is correctly situated on the first key segment.

```
Show[{gkey, gdisk[1, 1]}]
```



This `Manipulate` shows the various possible positions of the ring.

```
Manipulate[Show[{gkey, gdisk[i, j]}], {i, 1, 21, 1},
 {j, 1, 16, 1}, SaveDefinitions → True]
```



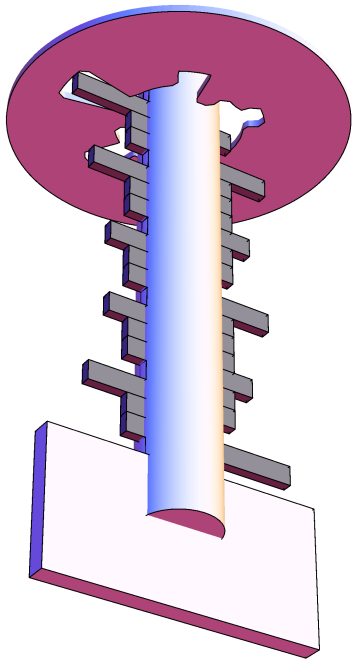
Not only can we adjust the position and orientation of the ring, but we can turn the whole puzzle with the mouse. Of course, this is not very realistic: you can slide the ring right through all the teeth on the key!

▪ *Routine to Display Puzzle*

And now here is the function to display the whole puzzle. We specify i and j, indices giving the position of the disk and its rotation.

```
showPuzzle[{i_Integer, j_Integer}, opts___] :=
 Show[{gkey, gdisk[i, j]}, opts]

showPuzzle[{20, 1}, ViewPoint -> {-2.35, 1.54, -1.87}]
```



□ Create Flat Maze Representation of Puzzle

▪ *Building the Maze*

We now turn our attention to creating a maze that models the same behavior as the puzzle. The illegal positions represent choices of $\{i, j\}$ such that the j^{th} rotation of the disk cannot fit on the key at position i, because either the tooth above or the tooth below is too tall compared to the cut in the disk above or below, respectively. We use 0 and 1 to mark allowed and forbidden locations.

```
maze = Table[
  If[key[[1, i]] > disk[[j]] ||
    key[[2, i]] > disk[[Mod[j + 7, 16] + 1]] , 0, 1],
  {j, Length[disk]}, {i, Length[key[[1]]]}];
maze // MatrixForm
```

$$\begin{pmatrix}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{pmatrix}$$

```
Dimensions[maze]
```

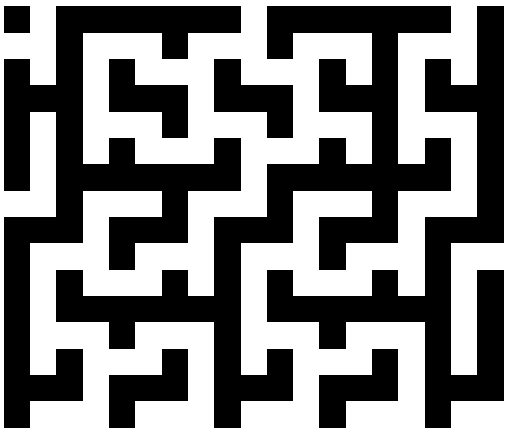
```
{16, 21}
```

In the first column we have all 1s, so all rotations of the disk are allowed there. But there are only two 1s in the next column. Only two orientations of the disk allow it to be slid over the teeth there; other orientations are forbidden (0s).

■ *Display the Flat Maze*

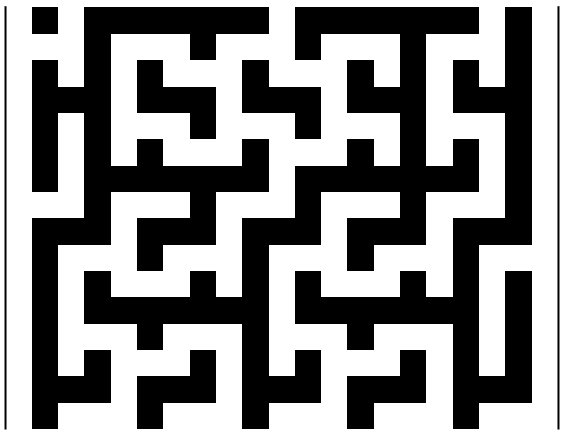
It would be much more convenient to see the maze as paths and walls. Let us create a list of pairs of `GrayLevel` graphics directives and `Rectangle` objects. Conveniently enough, `GrayLevel[0]` means black and `GrayLevel[1]` means white.

```
Show[
 Graphics[
  Flatten[
   Table[{GrayLevel[maze⟦j, i⟧],
     Rectangle[{i, j}, {i + 1, j + 1}]}, {i, 1, 21},
    {j, 1, 16}]]]]
```



To distinguish the white columns at the sides from the background, we draw a line at the beginning and end of the puzzle.

```
flatMaze :=
 Graphics[
  Flatten[
   {Table[{GrayLevel[maze⟦j, i⟧],
      Rectangle[{i, j}, {i + 1, j + 1}]}, {i, 1, 21},
     {j, 1, 16}], GrayLevel[0], Line[{{1, 1}, {1, 17}}],
    Line[{{22, 1}, {22, 17}}]}]]
Show[flatMaze]
```



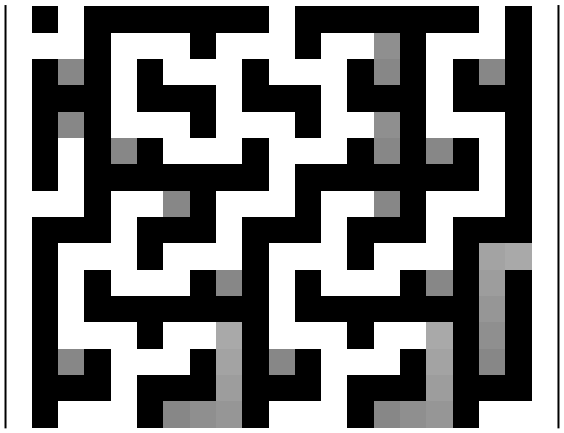
■ Procedure to Identify Dead-End Locations

It is easy to check whether a given allowed position is a dead end: at least three neighboring positions are not 1. Obviously if a position is part of the path through the maze, it must have at least a way to get there and a way to continue on—two allowed neighboring positions. The function `checkDeadEnd` counts the number of illegal and blocked neighbors. If there are three or more, it labels the current square as blocked by putting a number not equal to 1 in that position in the array. (The sign of a legal but not useful position is any number between 0 and 1.) But now more cells are seen to be dead ends if their neighbors are illegal *or* blocked. The blocking code number assigned starts at 0.5 but is always a little closer to 1 than the smallest blocking code number of the neighbors. This way any cell marked with 0 is illegal, any cell marked 0.5 has at least three illegal neighbors, and any cell marked with a code between 0.5 and 1 has at least three blocked or illegal neighbors, and has `blockingcode` a little greater than any of them (but never quite equal to 1).

```
checkDeadEnd[maze0_] :=
  Module[{maze = maze0, i, j, blockedneighbors,
    blockingcode},
   Do[
    If[maze[[j, i]] ≠ 1, Continue[]];
    blockedneighbors = 0; blockingcode = 0.5;
    If[maze[[Mod[j - 2, 16] + 1, i]] < 1, blockedneighbors++;
     blockingcode = Max[blockingcode,
       maze[[Mod[j - 2, 16] + 1, i]] ]];
    If[maze[[Mod[j, 16] + 1, i]] < 1, blockedneighbors++;
     blockingcode = Max[blockingcode,
       maze[[Mod[j, 16] + 1, i]] ]];
    If[maze[[j, i - 1]] < 1, blockedneighbors++;
     blockingcode = Max[blockingcode, maze[[j, i - 1]]]];
    If[maze[[j, i + 1]] < 1 , blockedneighbors++;
     blockingcode = Max[blockingcode, maze[[j, i + 1]]]];
    If[blockedneighbors ≥ 3,
     maze[[j, i]] = (1 + 15 blockingcode) / 16],
    {i, 1, 20}, {j, 1, 16}];
   maze];
```

Notice how this fills in many dead-end paths immediately, indicating some (but not all!) of the undesirable paths in gray.

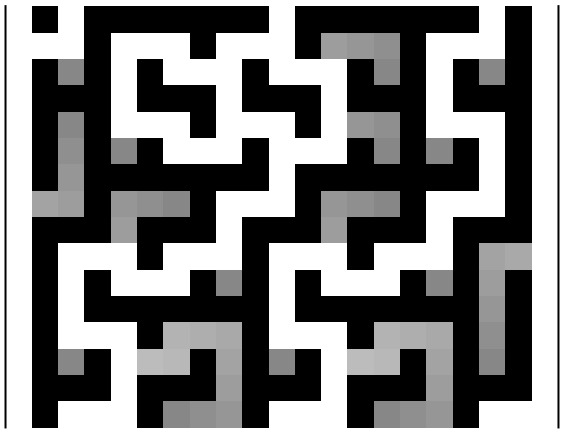
```
maze = checkDeadEnd[maze]; Show[flatMaze]
```



■ *Apply checkDeadEnd Repeatedly to Mark All Dead-End Paths*

In order to find *all* dead-end paths we need to apply the function repeatedly until no more changes occur. This is precisely what `FixedPoint` does. We hand it the function to apply and the initial state, and it keeps applying the function until done.

```
maze = FixedPoint[checkDeadEnd, maze];
Show[flatMaze]
```



Our blocking codes in `checkDeadEnd` have guaranteed gray shading for dead-end paths, starting with 0.5 and getting lighter as we move away from the cul-de-sac.

We still have 0 for illegal positions, and nonzero for allowed positions, but now any legal but undesirable position is marked by a nonzero number less than 1. Ones remain only on the *desirable* legal positions:

blocking code	meaning
`0.0`	illegal position
`[0.5, 1)`	legal, undesirable position
`1.0`	legal, desirable position

This makes the main path through the maze immediately obvious. In fact, there is only one place where alternate viable paths exist.

However, since the disk can be turned past 16 on to 1, the maze we have drawn wraps around vertically. It would make more sense to display it wrapped around a cylinder, and that is what we do next.

□ Create Cylindrical Maze, Turned Appropriately

▪ *The cylMaze Routine*

Now create a 3D object consisting of a list of `GrayLevel` directives and `Polygon` objects wrapped around a cylinder. We put in an argument defining how far around it should be turned, and memorize the results for each turn specified so as not to have to recalculate each time the function is called. `ListAnimate` stores its results but `Animate` does not.

```
cylMaze[turn_Integer] :=
 cylMaze[turn] = Module[{i, j, j0, fg},
   Graphics3D[
    Flatten[Table[{GrayLevel[maze[[j0, i]]],
       j = Mod[j0 + 3 - turn, 16];
       Polygon[{{3 Cos[2/16 π (j - 1)], i, 3 Sin[2/16 π (j - 1)]},
          {3 Cos[2/16 π (j - 1)], i + 1, 3 Sin[2/16 π (j - 1)]},
          {3 Cos[2 π j/16], i + 1, 3 Sin[2 π j/16]},
          {3 Cos[2 π j/16], i, 3 Sin[2 π j/16]}}]}, {i, 1, 21},
      {j0, 1, 16}]], ViewPoint → {2.464, 1.734, 1.54},
    Boxed → False, Lighting → {{"Ambient", White}}]]
```

`Animate` does not store its results in the notebook from one session to another (unlike `ListAnimate`, which does), so to see the result of this command you need to evaluate the input cells up to this point.

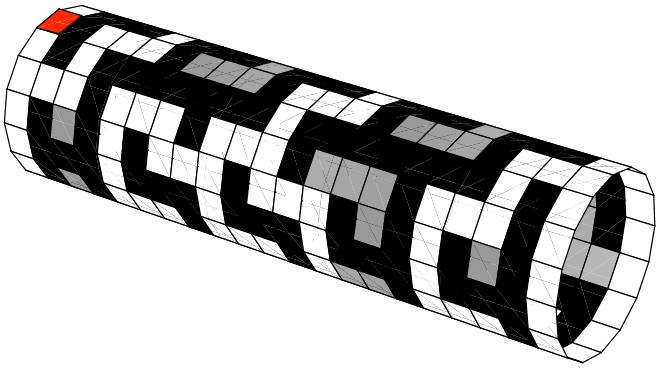
```
SetOptions[Animate, AnimationRepetitions → 1,
  AnimationDirection → ForwardBackward,
  DisplayAllSteps → True];
Animate[Show[cylMaze[turn]], {turn, 1, 16, 1}]
```

■ *The showSpotCyl Routine*

Why not put a red square somewhere on (or above) a specified {*i*, *j*} square on the cylinder? It could indicate our position as we walk through the maze.

```
showSpotCyl[{i_, j_}, opts___] :=
 Show[cylMaze[j - 1],
  Graphics3D[
   {Hue[0],
    Polygon[{{3.05 Cos[2/16 π (4 - 1 + 0)], i + 0,
       3.0` Sin[2/16 π (4 - 1 + 0)]},
      {3.05 Cos[2/16 π (4 - 1 + 0)], i + 1 + 0,
       3.05 Sin[2/16 π (4 - 1 + 0)]},
      {3.05 Cos[2/16 π (4 + 0)], i + 1 + 0, 3.05 Sin[2/16 π (4 + 0)]},
      {3.05 Cos[2/16 π (4 + 0)], i + 0, 3.05 Sin[2/16 π (4 + 0)]}}]},
  opts]
```

```
showSpotCyl[{1, 1}, ViewPoint → {2.464`, 1.734`, 1.54`},
 Lighting → {{"Ambient", White}}]
```



More on this later!

□ Solve Puzzle and Display Solution on Flat Maze

▪ *The findSoln Routine*

It is time to solve the puzzle. We start with $\{i, j\} = \{1, 1\}$ and arbitrarily try to move left, counterclockwise, clockwise, or right, in that order. Only moves to squares marked by a 1 in the maze array are considered. The moves are added to a list that is returned by the function, but at each move the new position is shown by a red square superimposed on the flat maze and a text explanation is added. Notice the command `Break[]` if no further move is possible, but this does not happen here.

```
findSoln := Module[{ans, lastmove = "right", i = 1, j = 1},
   ans = {{i, j}};
   While[i < 21, If[lastmove ≠ "left" && maze[[j, i + 1]] == 1,
     i++; lastmove = "right",
     If[lastmove ≠ "ccw" && maze[[Mod[j - 2, 16] + 1, i]] == 1,
      j = Mod[j - 2, 16] + 1; lastmove = "cw",
      If[lastmove ≠ "cw" && maze[[Mod[j, 16] + 1, i]] == 1,
       j = Mod[j, 16] + 1; lastmove = "ccw",
       If[lastmove ≠ "right" && maze[[j, i - 1]] == 1, i--;
        lastmove = "left", Print["No move ", {i, j}];
        Break[]]]]]; AppendTo[ans, {i, j}];
    Show[flatMaze,
     Graphics[{Hue[0], Rectangle[{i + 0.1`, j + 0.1`},
        {i + 1 - 0.1`, j + 1 - 0.1`}]}],
     PlotLabel →
      Style[Row[{"Moved ", lastmove, " to ", {i, j}}],
       FontFamily → "Times", FontSize → 12]]]; ans];

soln = findSoln;
```

■ *The Solution*

```
soln
```

```
{{1, 1}, {1, 16}, {1, 15}, {2, 15}, {3, 15}, {3, 16}, {3, 1},
 {4, 1}, {5, 1}, {5, 2}, {5, 3}, {5, 4}, {4, 4}, {3, 4},
 {3, 5}, {3, 6}, {3, 7}, {4, 7}, {5, 7}, {5, 6}, {6, 6},
 {7, 6}, {7, 7}, {8, 7}, {9, 7}, {9, 8}, {9, 9}, {10, 9},
 {11, 9}, {11, 10}, {11, 11}, {12, 11}, {13, 11}, {13, 12},
 {13, 13}, {13, 14}, {12, 14}, {11, 14}, {11, 15}, {11, 16},
 {11, 1}, {12, 1}, {13, 1}, {13, 2}, {13, 3}, {13, 4}, {12, 4},
 {11, 4}, {11, 5}, {11, 6}, {11, 7}, {12, 7}, {13, 7},
 {13, 6}, {14, 6}, {15, 6}, {15, 7}, {16, 7}, {17, 7},
 {17, 8}, {17, 9}, {18, 9}, {19, 9}, {19, 10}, {19, 11},
 {19, 12}, {18, 12}, {17, 12}, {17, 13}, {17, 14}, {17, 15},
 {18, 15}, {19, 15}, {19, 16}, {19, 1}, {20, 1}, {21, 1}}
```

■ *The Solution + Delays*

An easy way to add a delay at the beginning and end of the animation is to repeat the first and last positions a few times.

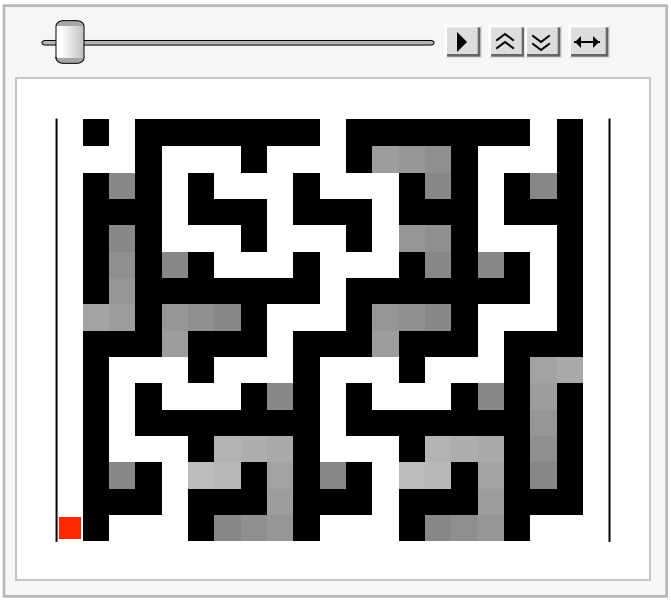
```
soln =
 Flatten[{Table[First@soln, {5}], soln,
   Table[Last@soln, {5}]}, 1]
```

```
{{1, 1}, {1, 1}, {1, 1}, {1, 1}, {1, 1}, {1, 1}, {1, 16},
 {1, 15}, {2, 15}, {3, 15}, {3, 16}, {3, 1}, {4, 1},
 {5, 1}, {5, 2}, {5, 3}, {5, 4}, {4, 4}, {3, 4}, {3, 5},
 {3, 6}, {3, 7}, {4, 7}, {5, 7}, {5, 6}, {6, 6}, {7, 6},
 {7, 7}, {8, 7}, {9, 7}, {9, 8}, {9, 9}, {10, 9}, {11, 9},
 {11, 10}, {11, 11}, {12, 11}, {13, 11}, {13, 12}, {13, 13},
 {13, 14}, {12, 14}, {11, 14}, {11, 15}, {11, 16}, {11, 1},
 {12, 1}, {13, 1}, {13, 2}, {13, 3}, {13, 4}, {12, 4},
 {11, 4}, {11, 5}, {11, 6}, {11, 7}, {12, 7}, {13, 7},
 {13, 6}, {14, 6}, {15, 6}, {15, 7}, {16, 7}, {17, 7},
 {17, 8}, {17, 9}, {18, 9}, {19, 9}, {19, 10}, {19, 11},
 {19, 12}, {18, 12}, {17, 12}, {17, 13}, {17, 14},
 {17, 15}, {18, 15}, {19, 15}, {19, 16}, {19, 1}, {20, 1},
 {21, 1}, {21, 1}, {21, 1}, {21, 1}, {21, 1}, {21, 1}}
```

■ *Display the Solution*

```
showFlat[{i_Integer, j_Integer}] :=
 Show[
  {flatMaze,
   Graphics[
    {Hue[0], Rectangle[{i + .1, j + .1},
      {i + 1 - .1, j + 1 - .1}]}]}]
```

```
SetOptions[ListAnimate, AnimationRepetitions → 1,
  AnimationDirection → ForwardBackward];
ListAnimate[showFlat /@ soln]
```



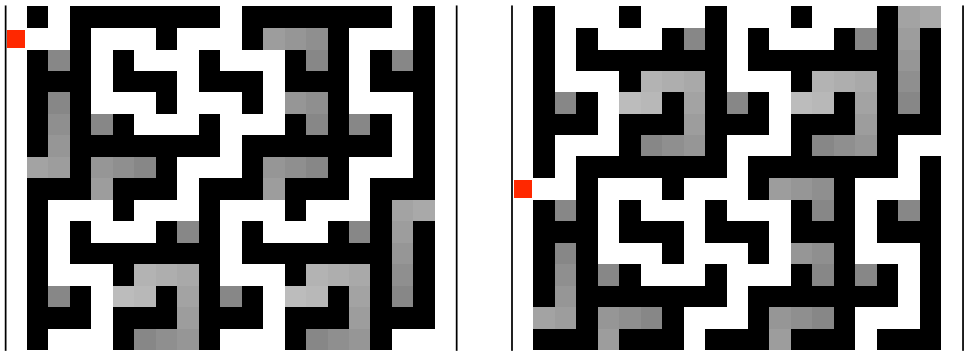
■ *Display It on a Sliding, Wrapping-Around Flat Maze*

Without going into a lot of detail here, the function `showFlatSliding` first shifts the y coordinates of all rectangles to display the value j in the middle (at position 8 out of 17) before showing the maze. Basically the red marker stays at the same height while the maze itself shifts up/down, wrapping around as needed.

```
showFlatSliding[{i_Integer, j_Integer}] :=
 Show[{Replace[flatMaze,
    Rectangle[{i1_, j1_}, {i2_, j2_}] :→
     Rectangle[{i1, Mod[j1 - j + 8, 16, 1]},
       {i2, Mod[j2 - j + 8, 16, 2]}], {2}],
   Graphics[
    {Hue[0], Rectangle[{i + .1, 8 + .1}, {i + 1 - .1, 9 - .1}]}]}]
```

Here are the two displays side by side for comparison.

```
GraphicsArray[{{showFlat[#], showFlatSliding[#]}}] & @
 {1, 15}
```



This command shows the solution using `ListAnimate` on the sliding flat maze. To keep a manageable file size, the results of `ListAnimate` have not been stored from this point on. The command takes about 30 seconds to run on a slow system.

```
ListAnimate[showFlatSliding /@ soln]
```

□ Display Solution on Cylindrical Maze

We need to map the function `showSpotCyl` onto the list `soln` and pass the result to `ListAnimate`. Run this command to see the result (about a minute on a slow system).

```
ListAnimate[showSpotCyl /@ soln]
```

No, that is not the best way after all. `ListAnimate` is showing only one premade image at a time, and only lets you drag the object to a different viewpoint in that one frame. Let us redo this using `Animate`. Now at any time you can pause the animation and experiment with adjusting the viewpoint (by clicking and dragging), then continue the action (click the **Play** triangle). The downside is that `Animate` runs more slowly than `ListAnimate`, since the frames are not precomputed. Also, you need to evaluate the commands defined up to this point; `ListAnimate` stores its results but `Animate` does not.

```
Animate[showSpotCyl[soln[[t]]], {t, 1, Length[soln], 1}]
```

It is best to run only one animation at a time so as not to slow down other animations or evaluations.

Apply Solution to Visual Puzzle

It is a little more time consuming to show this set of moves on the original key and disk puzzle. (Again, for `Animate`, you need to evaluate the commands up to this point.)

```
Animate[showPuzzle[soln[[t]]], {t, 1, Length[soln], 1}]
```

Putting It All Together

Solution Simultaneously Shown on Flat or Cylindrical Mazes while Moves Are Performed on the Puzzle Itself

I cannot resist showing different versions of the puzzle simultaneously in one animation, using `GraphicsGrid`. The fun here is in seeing exactly how the moves on the key/disk puzzle correspond to moves in the maze. `Animate` computes its results on the fly, so they cannot be stored and it takes time to generate the frames one after the other. To run a little faster you can replace `Animate` by `ListAnimate@Table`, which precomputes all the frames, keeping them in a `Table` (here a list of `GraphicsGrid` objects) that is passed to `ListAnimate`. Again, the price of this added speed during playback is some upfront computation time the first time the command is executed and the loss of interactive manipulability of the animated objects, which is probably a decent trade-off in this case.

```
Animate[GraphicsGrid[{{showFlatSliding[soln[[t]]]},
   {showPuzzle[soln[[t]], ImageSize → {300, 200},
     ViewPoint → {-2.7, 1.0, 1.7},
     ViewVertical → {-0.5, 1.6, 0.2}]}}],
 {t, 1, Length[soln], 1}]
```

```
Animate[GraphicsGrid[{{showSpotCyl[soln[[t]]],
    showPuzzle[soln[[t]], ImageSize → {300, 200},
     ViewPoint → {-2.66, 1.32, 1.61},
     ViewVertical → {-0.38, 1.68, 0.08}]}}],
 {t, 1, Length[soln], 1}]
```

These run fairly slowly; imagine what including all three is like! I show all three together here mostly to show how easy it is to place graphs where you want them.

```
Animate[GraphicsGrid[{
   {showFlatSliding[soln[[t]]], showPuzzle[soln[[t]]]},
   {showSpotCyl[soln[[t]]], SpanFromAbove}}],
 {t, 1, Length[soln], 1}]
```

Finally, here is the triple animation precomputed and shown with `ListAnimate`. It takes time to execute (three minutes on a slow system) and the 3D graphics cannot be rotated with the mouse, but once computed it runs at a reasonable speed.

```
ListAnimate@Table[GraphicsGrid[{
     {showFlatSliding[soln[[t]]], showPuzzle[soln[[t]]]},
     {showSpotCyl[soln[[t]]], SpanFromAbove}},
   ImageSize → {500, Automatic}], {t, 1, Length[soln], 1}]
```

Conclusion

This puzzle was fun to do, and used some interesting programming tricks. I particularly liked building up the visualization piece by piece, using *Mathematica* to generate everything from simple lists of key and disk cut heights/depths. And one great "Eureka!" moment was the realization that the puzzle was basically equivalent (isomorphic!) to a rectangular grid maze on the surface of a cylinder.

We might be able to think of a few more questions, though, such as:

- How were the tooth heights of the puzzle chosen?
- Is there any other choice of key and disk data (tooth heights and disk cut heights) that would result in a maze this complicated? I imagine that most choices would give either impossible mazes (the disk cannot move past a certain point) or trivially easy ones (the disk slides off).
- Might some other choice of key and disk data result in a *better* maze?
- We can generate a maze from any set of key and disk data. Is the converse true, can we take an arbitrary cylindrical maze and find key and disk heights for it? (My guess is "no.")

Ah! More programs to write, more ideas to try out!

Acknowledgments

I thank my colleagues at Southern Adventist University who have encouraged me, the folks at Wolfram Research who have occasionally helped me, and Claryce, who has put up with me in all my most puzzling moods.

References

[1] "Bits and Pieces." (March 11, 2010) www.bitsandpieces.com/product.asp?pn=42735.

[2] O. van Deventer. "OskarPuzzle's YouTube Channel." (March 11, 2010) www.youtube.com/user/OskarPuzzle.

K. E. Caviness, "Maze for Free the Key Puzzle," *The Mathematica Journal*, 2011.
dx.doi.org/doi:10.3888/tmj.13–1.

About the Author

Ken Caviness teaches physics at Southern Adventist University, a small liberal arts university near Chattanooga, Tennessee. He holds a Ph.D. in physics (emphases in relativity and nuclear physics) from the University of Massachusetts at Lowell, and has taught math and physics in Rwanda, Texas, and Tennessee. His interests include both computer and human languages (including the planned language Esperanto). He has used *Mathematica* since Version 1, both professionally and for recreational programming.

Kenneth E. Caviness
Physics Department
Southern Adventist University
P.O. Box 370, Collegedale, TN 37315-0370
caviness@southern.edu