Why measuring pi by counting pixels doesn't work

Jim Hall, MS

Technically We Write

ABSTRACT

This paper explores a naive approach to "measure" the value of pi by counting pixels on a screen to approximate the circumference. This approach does not yield the correct value of pi. This paper shows the flaws in the approach and why the calculated value is similar to but different from the pi=4 estimation using a square to approximate the circumference. The paper concludes by demonstrating another program that counts pixels inside a circle to estimate the area, which results in a more accurate calculation of pi.

1. Introduction

I used to write for Opensource.com, and every year on Pi Day (March 14, or 3/14) various authors would write at least one article about π. Several years ago, I wrote an article about how to calculate the value of π by writing a program to draw a circle to the screen, "measuring" the circumference of the circle by counting the pixels, then calculating the value for π by dividing the circumference by the diameter:

$$C = 2\pi r = \pi d$$

the value of π may be calculated as:

$$\pi = \frac{C}{d}$$

I thought this would be a fun article to write. I didn't think I would calculate the exact value of π, but I thought the answer would be "close enough" and I could move on to other articles.

I didn't expect the value to completely wrong. It wasn't even close. The calculation didn't improve by performing the same "measurement" at higher resolutions.

2. Method

To calculate this value of π, I wrote a short program using the Open Watcom C compiler on FreeDOS. I used a DOS program for this because using graphics mode in DOS is far simpler than manipulating graphics and counting pixels on other, more modern platforms like Linux. Entering graphics mode using Open Watcom C on DOS only requires only the `_setvideomode(`*mode*`)` function, using a value like `_VRES16COLOR` for the *mode* value to use 640×480 graphics with 16 colors, or `_SVRES256COLOR` for 800×600 or `_XRES256COLOR` for 1024×768 graphics with 256 colors. Using graphics on other platforms such as Linux requires initializing a graphics library with a long list of prerequisites, which results in much more complex code than I wanted to write for a simple "measure pi by counting pixels" article.

Drawing to the screen is also straightforward using DOS, like the `_setcolor(`*color*`)` function to set the drawing color to a 4-bit *i*RGB value, such as 7 for white or 15 for bright white, or `_ellipse(`*method*`, `*x1*`, `*y1*`, `*x2*`, `*y2*`)` function to draw an ellipse starting at *x1,y1* at the upper-left corner to *x2,y2* at the lower-right corner. Using `_GBORDER` for the *method* draws only the outline of the ellipse; using `_GFILLINTERIOR` draws a solid ellipse. To draw a circle, set the *x* distance to the same as the *y* distance; on a 640×480 screen, `_ellipse(_GBORDER, 0, 0, 479, 479)` draws the outline of a circle with diameter 480 starting at the upper-left corner of the screen.

To estimate a "measurement" of the circumference, the program iterated through all coordinates on the screen, and used the `_getpixel(`*x*`, `*y*`)` function to retrieve the color at each *x,y* coordinate. If the pixel had a color, the program counted it as the border of the circle. The total pixel count was used as the circumference, and the width of the circle was the diameter. Thus, the program calculated a value of π by dividing the circumference by the diameter:

$$\pi = \frac{C}{d}$$

3. Result

At 640×480 resolution, the program counted 1356 pixels for the "circumference" of a circle with diameter 480 pixels. This is a calculated value of:

$$\pi = \frac{1356}{480} = 2.825$$

Using a higher resolution at 800×600 pixels, the program counted 1696 pixels for the "circumference" and thus calculated:

$$\pi = \frac{1696}{600} \approx 2.82666666667$$

At the maximum DOS resolution of 1024×768, the program counted 2172 pixels for the "circumference" for a calculated value of:

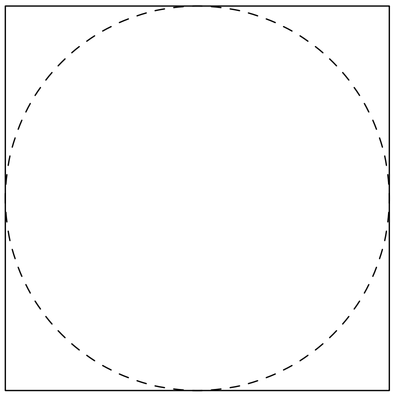
$$\pi = \frac{2172}{768} = 2.828125$$

A program to count pixels to calculate π:

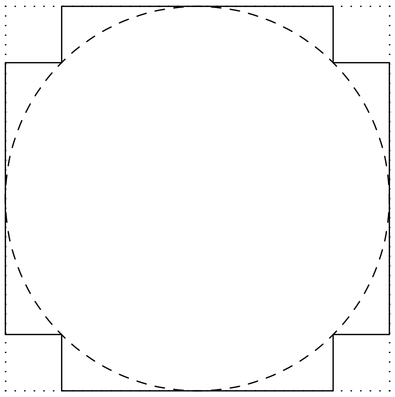
```
#include <stdio.h>
#include <graph.h>

int main()
{
  int x,y;
  unsigned long count = 0;

  if (_setvideomode(_XRES256COLOR) == 0) {
    return 1;
  }

  _setcolor(7);
  _ellipse(_GBORDER, 0, 0, 767, 767);

  for (x = 0; x < 768; x++) {
    for (y = 0; y < 768; y++) {
      if (_getpixel(x, y) > 0) {
        count++;
      }
    }
  }

  _setvideomode(_DEFAULTMODE); /* text */

  printf("pi = C/d = %ld/768 = %f0,
    count, (float)count / 768.0);

  return 0;
}
```

4. Analysis

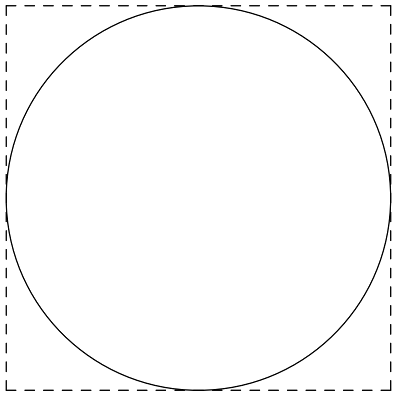
Counting pixels to "measure" the circumference of a circle is a naive approach. The issue is similar to the $\pi = 4$ estimation, which starts by drawing a square as a rough approximation of a circle:



To more closely approximate the circle, the estimation "folds" down each corner of the square to just touch the circle:



The naive assumption with this estimation is that, as the "steps" become infinitesimally small, the perimeter of the drawn lines approaches the actual circumference of the circle. However, the x and y components of each "step" still add up to the perimeter of the original square. The "circumference" of this estimation remains the perimeter of the original square.



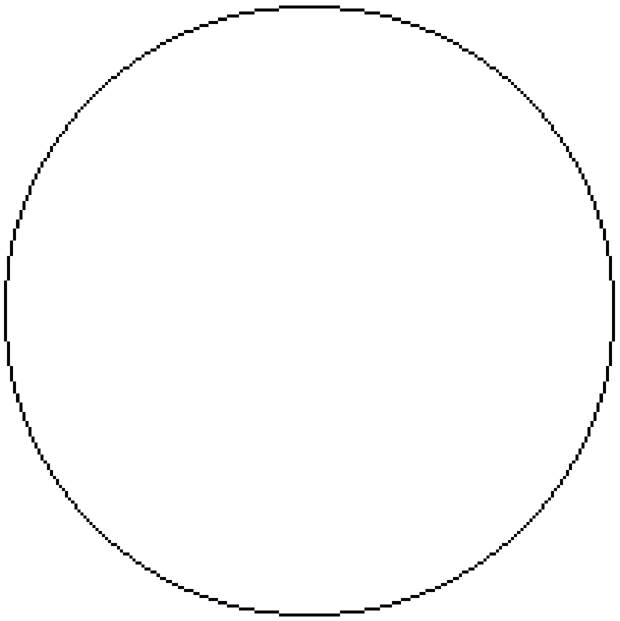
For a unit circle, each side of the bounding square has length 2, and a perimeter of 2×4 or 8. Thus, the resulting calculation of π is:

$$\pi = \frac{C}{d} = \frac{8}{2} = 4$$

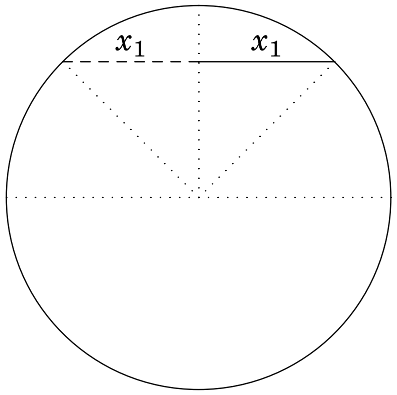
Counting pixels

The estimation using pixels to draw the outline of a circle, then counting the pixels for the "circumference" is similar, but made worse by the fact that pixels are points, and do not have both an x and y dimension. Each pixel is therefore an estimate in one dimension, where the "outline" method is an estimate in two dimensions: x and y.

Calculating the expected value by counting pixels requires understanding how the `_ellipse` function draws circles using pixels. The diagram shows a low resolution circle drawn in 320×200 graphics mode, so the cropped image is 200 pixels on each side. Close examination shows that the drawing algorithm places pixels horizontally at the top and bottom of the circle, and vertically on the left and right. Between these sides, the drawing algorithm places pixels diagonally. This placement is significant because at a midpoint on the arc between, for example, "left" and "top," the drawing algorithm does not place more than one pixel in a vertical arrangement.



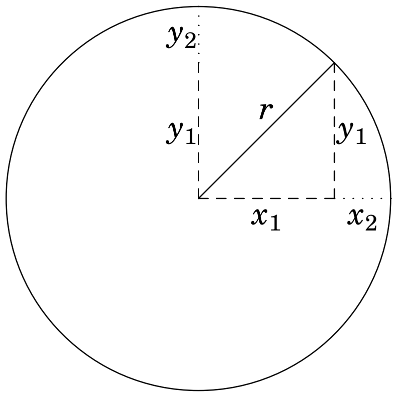
This allows calculation of the expected number of pixels from the midway point to the top by calculating the unit x distance from this midpoint to the center line of the circle, to calculate a "slice" of the circle. Because pixels are points, the integer value of this distance should equal the number of pixels that the drawing algorithm would generate for this arc.



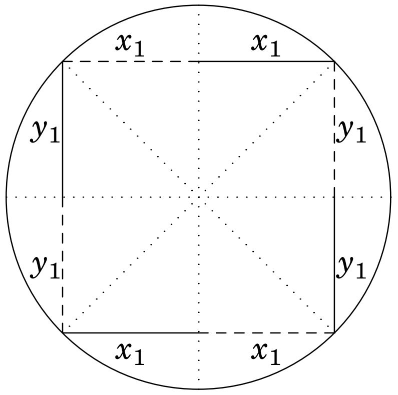
Further, because the circle is symmetrical, and because this is a one-eighth "slice" of the circle, multiplying the value by 8 should equal the total number of pixels produced by the drawing algorithm.

The distance

In a circle of radius r, the midpoint an arc is at the location where $x_1 = y_1$. This is also the angle $\theta = \pi/4$. For example:



Only the x_1 component is interesting for this calculation. Because $x_1 = y_1$, the value x_1 should be equal to one-eighth the "pixel" count for the circle.



The circle drawn at the highest possible DOS resolution of 1024×768 has a diameter of 768 pixels, or a radius of 384. Thus the length of x_1 can be calculated:

$$x_1 = r \cos \frac{\pi}{4} = 384 \times \cos \frac{\pi}{4}$$

This calculation is for a one-eighth slice of the circle. Multiplying times 8 gives the final result:

$$8 \times x_1 = 8 \times 384 \times \cos \frac{\pi}{4}$$

This calculation results in the floating point value 2172.23203180507, approximate. Because pixels are points and counting pixels is an integer value, the final expectation value of the number of pixels in a drawn circle is the integer value 2172. This matches the number of pixels counted by the DOS program as the circumference, when drawn in 1024×768 graphics resolution.

Further, if the diameter is 768, then we can complete the calculation for this expected approximation of π as:

$$\pi = \frac{C}{d} = \frac{2172}{768} = 2.828125$$

This is the same calculation of π as used in the DOS program. But at least it's nice to know where the values came from.

5. Discussion

Counting pixels for the circumference of a circle is a naive way to calculate a value for π, but a slightly different method generates much more accurate results: counting pixels for the area of a circle. Using the area in this calculation is a better approximation, because as the pixels become smaller at higher resolutions, the pixels that fill the area approaches the area.

Writing a program to calculate pi in this way requires only a minor modification to the original program: instead of `_GBORDER` for the *method* to draw the circle, use `_GFILLINTERIOR`. That parameter changes the behavior of the `_ellipse` function to fill the interior of the circle. With that change, counting pixels presents an estimate of the area. Using this as an estimate of the area, the program can calculate π using:

$$A = \pi r^2$$

and:

$$\pi = \frac{A}{r^2}$$

Working in 1024×768 graphics mode, the new version of the program counts 463488 pixels in the area of a circle with diameter 768. Dividing the count by the radius (384) twice to effect the square of the radius without overflowing variable limites, the program calculates the value of π as 3.143229, which is not bad for counting pixels to calculate π.

Updated program to count pixels to calculate π:

```
#include <stdio.h>
#include <graph.h>

int main()
{
  int x,y;
  unsigned long count = 0;

  if (_setvideomode(_XRES256COLOR) == 0) {
    return 1;
  }

  _setcolor(7);
  _ellipse(_GFILLINTERIOR, 0, 0, 767, 767);

  for (x = 0; x < 768; x++) {
    for (y = 0; y < 768; y++) {
      if (_getpixel(x, y) > 0) {
        count++;
      }
    }
  }

  _setvideomode(_DEFAULTMODE); /* text */

  /* 768/2 = 384 */
  printf("pi = A / r^2 = %ld / r^2 = %f0,
    count, (float)count/384.0/384.0);

  return 0;
}
```