Exploring Binary

# Print Precision of Dyadic Fractions Varies by Language

By Rick Regan January 30th, 2009

Interestingly, programming languages vary in how much precision they allow in printed floating-point fractions. You would think they'd all be the same, allowing you to print as many decimal places as you ask for. After all, a floating-point fraction is a dyadic fraction; it has as many decimal places as it has bits in its fractional representation.

Consider the dyadic fraction 5,404,319,552,844,595/$2^{53}$. Its decimal expansion is 0.59999999999999997779553950749686919152736663818359375, and its binary expansion is 0.10011001100110011001100110011001100110011001100110011. Both are 53 digits long. The ideal programming language lets you print all 53 decimal places, because all are meaningful. Unfortunately, many languages won't let you do that; they typically cap the number of decimal places at between 15 and 17, which for our example might be 0.59999999999999998.

I can guess why this is so. We view printed values as decimal numbers, not binary approximations. 0.59999999999999997779553950749686919152736663818359375 looks like 0.6, and in fact is what's stored in a double-precision floating-point variable that is assigned the value 0.6 (it's the closest double-precision approximation of 0.6). If you print this variable, you can live with it being $0.5\overline{9}$, or a finite version thereof. But all those digits after the last 9 look like garbage, and in fact they are with respect to 0.6.

Now change your perspective. Assume you care about binary approximations. You might want to see which dyadic fraction a decimal fraction rounds to. You might want to print a variable that has been assigned a dyadic fraction explicitly. You might want to know the value of a small negative power of two. In these cases, you'll want to see the entire decimal expansion, "garbage" and all.

Let's see how good various programming languages are at giving you this precision.

## Summary of Results

I tested the printing precision of eight different languages, some on both Windows and Linux, for a total of twelve environments. I determined the maximum precision each language allows when printing a double-precision floating-point value (AKA double). I picked two cases for each, the smallest and largest (positive) dyadic fractions that could be printed in their entirety.

The *smallest* fraction is a negative power of two. The smallest negative power of two that fits in a double is $2^{-1074}$. In decimal, it is 323 0s followed by 751 other digits. In binary, it is 1073 0s followed by a 1 (of course all those digits don't fit in a double — they are implied by the exponent).

The *largest* fraction is of the form $(2^n – 1)/2^n$, or $1 – 2^{-n}$. For a double, the largest value possible is $1 – 2^{-53}$. In decimal, it is 15 9s followed by 38 other digits. In binary, it is 53 1s.

Here's a summary of what I found (striked out text reflects updated versions — see below for details):

**Longest Dyadic Fractions Printed in Full Precision**

| Language | OS | Smallest Fraction | Largest Fraction |
|---|---|---|---|
| GCC (C) | Linux | $2^{-1074}$ | $1 – 2^{-53}$ |
| Visual C++ | Windows | $2^{-24}$ | $1 – 2^{-17}$ |
| Java | Windows | $2^{-24}$ | $1 – 2^{-16}$ |
| Visual Basic | Windows | $2^{-21}$ | $1 – 2^{-15}$ |
| PHP | Windows | ~~$2^{-40}$~~ $2^{-53}$ | ~~$1 – 2^{-40}$~~ $1 – 2^{-53}$ |
| JavaScript (Firefox) | Windows | $2^{-100}$ | $1 – 2^{-53}$ |
| JavaScript (IE) | Windows | $2^{-20}$ | $1 – 2^{-16}$ |
| VBScript (IE) | Windows | $2^{-21}$ | $1 – 2^{-15}$ |
| ActivePerl | Windows | $2^{-24}$ | $1 – 2^{-17}$ |
| Python | Windows | ~~$2^{-24}$~~ $2^{-1074}$ | ~~$1 – 2^{-17}$~~ $1 – 2^{-53}$ |
| Perl | Linux | $2^{-1074}$ | $1 – 2^{-53}$ |
| Python | Linux | $2^{-117}$ | $1 – 2^{-53}$ |

The GNU Compiler Collection (GCC), and specifically its C compiler, is what I'll call the gold standard. It prints all dyadic fractions to full precision. Perl on Linux, which presumably is built using GCC and glibc, does the same.

JavaScript in Firefox is curious. It prints the largest double fraction, $1 - 2^{-53}$, but only prints powers of two up to $2^{-100}$. It seems like an arbitrary maximum, perhaps due to a buffer size limit. PHP is similar. It picks an arbitrary maximum of ~~40~~ 53 decimal places, regardless of the fraction. Python on Linux also has an interesting limit — it prints powers of two up to $2^{-117}$.

What follows are the details of how I determined the values in the table. For each value, I show two pieces of code: one that prints the longest fraction, and one that fails to print the next longer fraction. In other words, I determine the maximum by going beyond it.

## 1. GCC C (Linux)

### Smallest Fraction

This code prints all 1,074 decimal places of $2^{-1074}$:

```
#include "stdio.h"
int main()
{
/* Print 2^-1074 */
double dyadic = /* Compute as (2^-64)^16 * 2^-50 to break up */
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.00000000000000088817841970012523233890533447265625;
printf ("%1.1074f",dyadic);
}
```

This code does not print $2^{-1075}$. This is expected. $2^{-1075}$ underflows to 0 in a double, so 1075 decimal places of 0 are printed instead:

```
#include "stdio.h"
int main()
{
/* Print 2^-1075 */
double dyadic = /* Compute as (2^-64)^16 * 2^-51 to break up */
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.000000000000000444089209850062616169452667236328125;
printf ("%1.1075f",dyadic);
}
```

## Largest Fraction

This code prints all 53 decimal places of 1 – $2^{-53}$:

```
#include "stdio.h"
int main()
{
/* Print 1 - 2^-53 */
double dyadic =
   0.99999999999999988897769753748434595763683319091796875;
printf ("%1.53f",dyadic);
}
```

This code does not print 1 – $2^{-54}$; it prints 1 followed by 54 decimal places of 0s (the compiler, during decimal to binary conversion, rounds this "halfway" case up to 1.0):

```
#include "stdio.h"
int main()
{
/* Print 1 - 2^-54 */
double dyadic =
   0.999999999999999944488848768742172978818416595458984375;
printf ("%1.54f",dyadic);
}
```

## 2. Visual C++ (Windows)

### Smallest Fraction

This code prints all 24 decimal places of $2^{-24}$:

```
#include "stdio.h"
int main()
{
/* Print 2^-24 */
double dyadic = 0.000000059604644775390625;
printf ("%1.24f",dyadic);
}
```

This code does not print $2^{-25}$, but instead 0.0000000298023223876953130:

```
#include "stdio.h"
int main()
{
/* Print 2^-25 */
double dyadic = 0.0000000298023223876953125;
printf ("%1.25f",dyadic);
}
```

### Largest Fraction

This code prints all 17 decimal places of $1 - 2^{-17}$:

```
#include "stdio.h"
int main()
{
/* Print 1 - 2^-17 */
double dyadic = 0.99999237060546875;
printf ("%1.17f",dyadic);
}
```

This code does not print $1 - 2^{-18}$, but instead 0.999996185302734380:

```
#include "stdio.h"
int main()
{
/* Print 1 - 2^-18 */
double dyadic = 0.999996185302734375;
printf ("%1.18f",dyadic);
}
```

## 3. Java (Windows)

### Smallest Fraction

This code prints all 24 decimal places of $2^{-24}$:

```
class precision {
   /* Print 2^-24 */
   public static void main(String[] args) {
      double dyadic = 0.000000059604644775390625;
      System.out.printf("%1.24f\n",dyadic);
    }
}
```

This code does not print $2^{-25}$, but instead 0.0000000298023223876953120:

```
class precision {
   /* Print 2^-25 */
   public static void main(String[] args) {
      double dyadic = 0.0000000298023223876953125;
      System.out.printf("%1.25f\n",dyadic);
    }
}
```

### Largest Fraction

This code prints all 16 decimal places of $1 - 2^{-16}$:

```
class precision {
   /* Print 1 - 2^-16 */
   public static void main(String[] args) {
      double dyadic = 0.9999847412109375;
      System.out.printf("%1.16f\n",dyadic);
```

```
    }
}
```

This code does not print 1 – $2^{-17}$, but instead 0.99999237060546880:

```
class precision {
   /* Print 1 - 2^-17 */
   public static void main(String[] args) {
      double dyadic = 0.99999237060546875;
      System.out.printf("%1.17f\n",dyadic);
    }
}
```

## 4. Visual Basic (Windows)

### Smallest Fraction

This code prints all 21 decimal places of $2^{-21}$:

```
Module Precision
    'Print 2^-21
    Sub Main()
        Dim dyadic As Double
        dyadic = 0.000000476837158203125
        Console.WriteLine("{0:F21}", dyadic)
    End Sub
End Module
```

This code does not print $2^{-22}$, but instead 0.0000002384185791015630:

```
Module Precision
    'Print 2^-22
    Sub Main()
        Dim dyadic As Double
        dyadic = 0.0000002384185791015625
        Console.WriteLine("{0:F22}", dyadic)
    End Sub
End Module
```

### Largest Fraction

This code prints all 15 decimal places of 1 – $2^{-15}$:

```
Module Precision
    'Print 1 - 2^-15
    Sub Main()
        Dim dyadic As Double
        dyadic = 0.999969482421875
        Console.WriteLine("{0:F15}", dyadic)
    End Sub
End Module
```

This code does not print 1 – $2^{-16}$, but instead 0.9999847412109380:

```
Module Precision
    'Print 1 - 2^-16
    Sub Main()
        Dim dyadic As Double
        dyadic = 0.9999847412109375
        Console.WriteLine("{0:F16}", dyadic)
    End Sub
End Module
```

## 5. PHP (Windows)

(Updated to reflect the fix for bug 47168, as tested in PHP 5.3.5. Curiously, the arbitrary limit still exists, albeit raised from 40 to 53 decimal places. I wrote a new bug to address this, but it doesn't look like they'll be fixing that one.)

### Smallest Fraction

~~This code prints all 40 decimal places of $2^{-40}$:~~

```
~~<?php~~
~~ /* Print 2^-40 */~~
~~ $dyadic = 0.0000000000009094947017729282379150390625;~~
~~ printf ("%1.40f",$dyadic);~~
~~?>~~
```

~~This code does not print $2^{-41}$; 0.0000000000004547473508864641189575195312 is printed instead:~~

```
~~<?php~~
~~ /* Print 2^-41 */~~
~~ $dyadic = 0.00000000000045474735088646411895751953125;~~
~~ printf ("%1.41f",$dyadic);~~
~~?>~~
```

This code prints all 53 decimal places of $2^{-53}$:

```
<?php
 /* Print 2^-53 */
 $dyadic = 0.00000000000000011102230246251565404236316680908203125;
 printf ("%1.53f",$dyadic);
?>
```

This code does not print $2^{-54}$; instead, it prints

0.00000000000000005551115123125782702118158340454101562.

```
<?php
 /* Print 2^-54 */
 $dyadic = 0.000000000000000055511151231257827021181583404541015625;
 printf ("%1.54f",$dyadic);
?>
```

## Largest Fraction

~~This code prints all 40 decimal places of $1 - 2^{-40}$:~~

```
~~<?php~~
~~ /* Print 1 - 2^-40 */~~
~~ $dyadic = 0.9999999999990905052982270717620849609375;~~
~~ printf ("%1.40f",$dyadic);~~
~~?>~~
```

~~This code does not print 1 – $2^{-41}$; instead, it prints 0.9999999999995452526491135358810424804688:~~

```
~~<?php~~
~~ /* Print 1 - 2^-41 */~~
~~ $dyadic = 0.99999999999954525264911353588104248046875;~~
~~ printf ("%1.41f",$dyadic);~~
~~?>~~
```

This code prints all 53 decimal places of 1 – $2^{-53}$:

```
<?php
 /* Print 1 - 2^-53 */
 $dyadic = 0.99999999999999988897769753748434595763683319091796875;
 printf ("%1.53f",$dyadic);
?>
```

This code does not print 1 – $2^{-54}$; it prints 1 followed by 53 decimal places of 0s:

```
<?php
 /* Print 1 - 2^-54 */
 $dyadic = 0.999999999999999944488848768742172978818416595458984375;
 printf ("%1.54f",$dyadic);
?>
```

## 6. JavaScript in Firefox (Windows)

### Smallest Fraction

This code prints all 100 decimal places of $2^{-100}$:

```
<script type="text/javascript">
  /* Print 2^-100 */
  var dyadic = /* Compute as 2^-50 * 2^-50 to break into 2 lines */
    0.00000000000000088817841970012523233890533447265625 *
    0.00000000000000088817841970012523233890533447265625;
  document.write (dyadic.toFixed(100));
</script>
```

This code does not print $2^{-101}$. Instead, it prints *nothing*. The Firefox error console gives the message “precision 101 out of range,” complaining about toFixed(101):

```
<script type="text/javascript">
  /* Print 2^-101 */
  var dyadic = /* Compute as 2^-50 * 2^-51 to break into 2 lines */
    0.00000000000000088817841970012523233890533447265625 *
    0.000000000000000444089209850062616169452667236328125;
  document.write (dyadic.toFixed(101));
</script>
```

This code is the same as above but with “toFixed(100).” It therefore can’t print $2^{-101}$. Instead it rounds the last two digits, ’25’, to ‘3’ (I didn’t want to paste a 100 digit decimal here):

```
<script type="text/javascript">
  /* Print 2^-101 */
  var dyadic = /* Compute as 2^-50 * 2^-51 to break into 2 lines */
    0.00000000000000088817841970012523233890533447265625 *
    0.000000000000000444089209850062616169452667236328125;
  document.write (dyadic.toFixed(100));
</script>
```

## Largest Fraction

This code prints all 53 decimal places of $1 - 2^{-53}$:

```
<script type="text/javascript">
   /* Print 1 - 2^-53 */
  var dyadic =
     0.99999999999999988897769753748434595763683319091796875;
  document.write (dyadic.toFixed(53));
</script>
```

This code does not print $1 - 2^{-54}$, but instead 1 followed by 54 decimal places of 0s:

```
<script type="text/javascript">
   /* Print 1 - 2^-54 */
  var dyadic =
     0.999999999999999944488848768742172978818416595458984375;
  document.write (dyadic.toFixed(54));
</script>
```

## 7. JavaScript in Internet Explorer (Windows)

### Smallest Fraction

This code prints all 20 decimal places of $2^{-20}$:

```
<script type="text/javascript">
  /* Print 2^-20*/
  var dyadic = 0.00000095367431640625;
  document.write (dyadic.toFixed(20));
</script>
```

This code does not print $2^{-21}$. Instead, it prints *nothing*. The Internet Explorer error console gives the message "Error: The number of fractional digits is out of range," complaining about toFixed(21):

```
<script type="text/javascript">
  /* Print 2^-21*/
  var dyadic = 0.000000476837158203125;
  document.write (dyadic.toFixed(21));
</script>
```

This code is the same as above but with "toFixed(20)." It therefore can't print $2^{-21}$. Instead it prints 0.00000047683715820313:

```
<script type="text/javascript">
  /* Print 2^-21*/
  var dyadic = 0.000000476837158203125;
  document.write (dyadic.toFixed(20));
</script>
```

### Largest Fraction

This code prints all 16 decimal places of $1 - 2^{-16}$:

```
<script type="text/javascript">
   /* Print 1 - 2^-16*/
  var dyadic =
     0.9999847412109375;
  document.write (dyadic.toFixed(16));
</script>
```

This code does not print 1 – $2^{-17}$, but instead 0.99999237060546870:

```
<script type="text/javascript">
   /* Print 1 - 2^-17 */
  var dyadic =
     0.99999237060546875;
  document.write (dyadic.toFixed(17));
</script>
```

## 8. VBScript in Internet Explorer (Windows)

### Smallest Fraction

This code prints all 21 decimal places of $2^{-21}$:

```
<script type="text/vbscript">
 'Print 2^-21
 dyadic = 0.000000476837158203125
 document.write(FormatNumber(dyadic,21))
</script>
```

This code does not print $2^{-22}$, but instead 0.0000002384185791015630:

```
<script type="text/vbscript">
 'Print 2^-22
 dyadic = 0.0000002384185791015625
 document.write(FormatNumber(dyadic,22))
</script>
```

### Largest Fraction

This code prints all 15 decimal places of 1 – $2^{-15}$:

```
<script type="text/vbscript">
 'Print 1 - 2^-15
 dyadic = 0.999969482421875
 document.write(FormatNumber(dyadic,15))
</script>
```

This code does not print 1 – $2^{-16}$, but instead 0.9999847412109380:

```
<script type="text/vbscript">
 'Print 1 - 2^-16
 dyadic = 0.9999847412109375
 document.write(FormatNumber(dyadic,16))
</script>
```

## 9. ActivePerl (Windows)

### Smallest Fraction

This code prints all 24 decimal places of $2^{-24}$:

```
# Print 2^-24
$dyadic = 0.000000059604644775390625;
printf "%1.24f",$dyadic;
```

This code does not print $2^{-25}$, but instead 0.0000000298023223876953130:

```
# Print 2^-25
$dyadic = 0.0000000298023223876953125;
printf "%1.25f",$dyadic;
```

### Largest Fraction

This code prints all 17 decimal places of 1 – $2^{-17}$:

```
# Print 1 - 2^-17
$dyadic = 0.99999237060546875;
printf "%1.17f",$dyadic;
```

This code does not print 1 – $2^{-18}$, but instead 0.999996185302734380:

```
# Print 1 - 2^-18
$dyadic = 0.999996185302734375;
printf "%1.18f",$dyadic;
```

## 10. Python (Windows)

(Updated to reflect Python 3.1.)

### Smallest Fraction

~~This code prints all 24 decimal places of $2^{-24}$:~~

```
~~# Print 2^-24~~
~~dyadic = 0.000000059604644775390625~~
~~print(format(dyadic,"1.24f"))~~
```

~~This code does not print $2^{-25}$, but instead 0.0000000298023223876953130:~~

```
~~# Print 2^-25~~
~~dyadic = 0.0000000298023223876953125~~
~~print(format(dyadic,"1.25f"))~~
```

This code prints all 1,074 decimal places of $2^{-1074}$:

```
#Print 2^-1074 (Compute as (2^-50)^21 * 2^-24 to break up)
dyadic = 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.00000000000000088817841970012523233890533447265625
dyadic *= 0.000000059604644775390625
print(format(dyadic,"1.1074f"))
```

## Largest Fraction

~~This code prints all 17 decimal places of 1 – $2^{-17}$:~~

```
~~# Print 1 - 2^-17~~
~~dyadic = 0.99999237060546875~~
~~print(format(dyadic,"1.17f"))~~
```

~~This code does not print 1 – $2^{-18}$, but instead 0.999996185302734380:~~

```
~~# Print 1 - 2^-18~~
~~dyadic = 0.999996185302734375~~
~~print(format(dyadic,"1.18f"))~~
```

This code prints all 53 decimal places of 1 – $2^{-53}$:

```
# Print 1 - 2^-53
dyadic = 0.99999999999999988897769753748434595763683319091796875
print(format(dyadic,"1.53f"))
```

## 11. Perl (Linux)

## Smallest Fraction

This code prints all 1,074 decimal places of $2^{-1074}$:

```
# Print 2^-1074
$dyadic = #Compute as (2^-64)^16 * 2^-50 to break up
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.00000000000000088817841970012523233890533447265625;
printf "%1.1074f",$dyadic;
```

This code does not print $2^{-1075}$. This is expected. $2^{-1075}$ underflows to 0 in a double, so 1075 decimal places of 0 are printed instead:

```
# Print 2^-1075
$dyadic = #Compute as (2^-64)^16 * 2^-51 to break up
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.0000000000000000000542101086242752217003726400434970855712890625*
0.000000000000000444089209850062616169452667236328125;
printf "%1.1075f",$dyadic;
```

### Largest Fraction

This code prints all 53 decimal places of 1 – $2^{-53}$:

```
# Print 1 - 2^-53
$dyadic = 0.99999999999999988897769753748434595763683319091796875;
printf "%1.53f",$dyadic;
```

This code does not print 1 – $2^{-54}$, but instead 1 followed by 54 decimal places of 0s:

```
# Print 1 - 2^-54
$dyadic = 0.999999999999999944488848768742172978818416595458984375;
printf "%1.54f",$dyadic;
```

## 12. Python (Linux)

### Smallest Fraction

This code prints all 117 digits of $2^{-117}$:

```
# Print 2^-117
# Compute as (2^-32)^3 * 2^-21 to break up
dyadic = 0.00000000023283064365386962890625
dyadic *= 0.00000000023283064365386962890625
dyadic *= 0.00000000023283064365386962890625
dyadic *= 0.000000476837158203125
print(format(dyadic,"1.117f"))
```

This code does not print $2^{-118}$. Instead, it prints the first 117 digits of $2^{-118}$ (it cuts off the last digit):

```
# Print 2^-118
# Compute as (2^-32)^3 * 2^-22 to break up
dyadic = 0.00000000023283064365386962890625
dyadic *= 0.00000000023283064365386962890625
dyadic *= 0.00000000023283064365386962890625
dyadic *= 0.0000002384185791015625
print(format(dyadic,"1.118f"))
```

### Largest Fraction

This code prints all 53 decimal places of 1 – $2^{-53}$:

```
# Print 1 - 2^-53
dyadic = 0.99999999999999988897769753748434595763683319091796875
print(format(dyadic,"1.53f"))
```

This code prints 1 – $2^{-53}$ with a 0 appended (the interpreter, during decimal to binary conversion, appears to be rounding this “halfway” case *down* to 1 – $2^{-53}$):

```
# Print 1 - 2^-54
dyadic = 0.999999999999999944488848768742172978818416595458984375
print(format(dyadic,"1.54f"))
```

## Should They All Be Like GCC?

Should all languages allow printing to maximum precision like GCC? I don’t see why not. Granted, there’s probably not much floating-point being done in scripting languages, but why not support it if it’s easy to implement? One unknown is compatibility. Are there programs that depend on the current imprecise output?

I tested the waters by submitting bug reports for Visual C++ and PHP. Microsoft took over ten months to consider my request but they acknowledged the problem and said they would change it in a future release (it seems to apply to all of Visual Studio by the way). On the other hand, PHP took only three hours to reject my request without a good explanation (*Update*: Rasmus reopened the bug.)

I don’t plan on submitting more bug reports at the moment, but if you decide to, let us know so we can track it.

## Other Languages and Environments

There are lots of things I didn’t try, like Fortran, Java on Linux, JavaScript in Opera, Safari, and Chrome, etc. If you are so inclined, please experiment and let us know what you find.

EB

## Related

- Quick and Dirty Floating-Point to Decimal Conversion
- Incorrect Floating-Point to Decimal Conversions
- Converting Floating-Point Numbers to Binary Strings in C
- Print Precision of Floating-Point Integers Varies Too

Get articles by RSS (What Is RSS?)

Get articles by e-mail

Numbers in computers / C, Code, Decimals, Exponents, Floating-point, Fractions, Java, JavaScript, PHP, Python

## 18 comments

1. **Rasmus**

   April 28, 2009 at 6:03 pm

   This is actually configurable in PHP. In your php.ini file, try this:

   precision=1024

   and run your PHP tests again.

2. **Rick Regan**

   April 29, 2009 at 8:21 am

   Hi Rasmus,

   Thanks for taking the time to comment.

   I tried changing php.ini as you suggested and I saw no difference.

   In the PHP source, in \ext\standard\formatted_print.c, there is this line of code: #define MAX_FLOAT_PRECISION 40. I've been assuming this was the culprit.

   P.S. Thanks for reopening the PHP bug!

3. **Christophe Poucet**

   April 29, 2009 at 9:34 am

   Don't forget dc and bc :).

4. **Kevin Gadd**

   April 29, 2009 at 9:44 pm

   If memory serves, PHP also has a hard-coded limit of 40 digits after the decimal place when serializing numbers using serialize().

This actually means that not only is precision lost when printing out doubles like the one you use in your example, but the precision may also be lost when serializing/unserializing them, which is painful.

This may have changed by now, though; I last researched it near the end of PHP4's lifetime. It was a surprise to me, to say the least (we were having weird issues with floating point numbers in some of our automated tests that turned out to be due to serialize()/unserialize().)

---

5. **Rick Regan**

April 30, 2009 at 10:34 am

Kevin,

I tried out serialize and unserialize — you're right, there are similar limitations.

$2^{-143}$ is the smallest negative power of two that PHP can serialize. However, it can only *unserialize* up to $2^{-40}$!

I will amend the bug report.

Here are the testcases:

This code works — it prints:
d:8.968310171678829253911869333055463240193676428009700939245237016894662929189507849514484405517578125E-44;

```
<?php
 //Serialize 2^-143
 $dyadic = //Compute as (2^-50)^2 * 2^-43 to break up
 0.00000000000000088817841970012523233890533447265625*
 0.00000000000000088817841970012523233890533447265625*
 0.0000000000001136868377216160297393798828125;
 echo serialize ($dyadic);
?>
```

This code fails — it prints:
d:4.4841550858394146269559346665277316200968382140048504696226185

08447331464594753924757242202758789062E-44;

```
<?php
 //Serialize 2^-144
 $dyadic = //Compute as (2^-50)^2 * 2^-44 to break up
 0.00000000000000088817841970012523233890533447265625*
 0.00000000000000088817841970012523233890533447265625*
 0.00000000000005684341886080801486968994140625;
 echo serialize ($dyadic);
?>
```

This code works — it prints:
0.0000000000009094947017729282379150390625

```
<?php
 //Serialize and unserialize 2^-40
 $dyadic = 0.0000000000009094947017729282379150390625;
 printf ("%1.40f",unserialize(serialize($dyadic)));
?>
```

This code fails — it prints:
0.00000000000045474735088646411895751953125

```
<?php
 //Serialize and unserialize 2^-41
 $dyadic = 0.00000000000045474735088646411895751953125;
 printf ("%1.41f",unserialize(serialize($dyadic)));
?>
```

(I did not try the “largest fraction” testcases but I’d imagine they’d have similar limitations.)

---

6. **Rick Regan**

April 30, 2009 at 1:10 pm

Christophe,

Since you mentioned it, I tried out bc and dc. Of course, they’re arbitrary-precision calculators, so they should handle any value.

bc
==

bc
scale=2000
dyadic=2^-2000
dyadic

(prints out 2^-2000 accurately, a 2000 digit number, bigger than can fit in a double.)

dc
==

dc
(I pasted the 2000 digit 2^-1000 value here)
p
(prints out 2^-2000 accurately)

---

7. **Rick Regan**

   February 14, 2010 at 3:43 pm

   I updated the table (Windows only for now) to reflect Python 3.1 (thanks to Mark Dickinson) — see https://www.exploringbinary.com/print-precision-of-floating-point-integers-varies-too/#comment-4273.

---

8. **Benjamin Rosseaux**

   August 10, 2010 at 8:27 am

   The ECMAScript spec (which JavaScript is based on) limits itself the methods toFixed etc. to a fixed "valid" precision/digitwidth value range. For example for toFixed, the ECMAscript 5th edition spec says: "If f 20, throw a RangeError exception." in "15.7.4.5 Number.prototype.toFixed (fractionDigits)" even if the engine uses dtoa.c or simliar good double-to-string routine implementation (like at my own ECMAscript 5th edition implementation BESEN) and could print plain technically more digits, bit the specification forbids it. For example my ECMAscript 5th edition implementation BESEN could technically the same precision/digitwidth range like GCC but due to the ECMAscript 5th edition specifications it is a range precheck there. Short: Sometimes are the too strict "standardized" language specifications itself the culprit for too low output precision and to less maximal output digit width.

---

9. **Rick Regan**

August 10, 2010 at 9:33 am

Benjamin,

Thanks for the detailed comment.

I took a look at section 15.7.4.5 (p. 155). It also says this:

> *An implementation is permitted to extend the behaviour of toFixed for values of fractionDigits less than 0 or greater than 20. In this case toFixed would not necessarily throw RangeError for such values.*

You'll also note that, while JavaScript in IE does in fact limit the number of digits to 20, JavaScript in Firefox limits the number of digits to 100. (And I contend that, since 100 is OK, why not 1,074?)

---

10. **Benjamin Rosseaux**

August 10, 2010 at 12:10 pm

Oh ok, I've overlook/miss it then. I've updated my own ECMAscript 5th Edition implementation now (Sourceforge project BESEN), which allows a higher valid precision/digitwidth value range. 🙂

BESEN is implemented in ObjectPascal, so I've ported&with-own-features-enhanced the dtoa.c algorithms from C to ObjectPascal, and additionally to dtoa.c's strtod I've wrote my own routine (which is based purely on arbitrary precision big integer operations without any FPU/SSE* operations) as second string-to-double-convert-routine, which give the exact results as dtoa.c's strtod, but works even on x64/AMD64 without SSE exceptions (because FreePascal generates on x64/AMD64 SSE* instruction instead of x87 FPU operartions and the floating point exception masking function in FreePascal isn't working correctly for SSE), so

BESEN is using to ObjectPascal ported dtoa.c's strtod on the 32-bit x86 target and on all other non-x86/x64/AMD64 targets and my own routine on the x64/AMD64 target, and for double-to-string converts it uses always the ported dtoa function with some enhancments/modifications/fixes from me, due to security as heap/buffer overflows and so on.

And the original double-converting routines from FreePascal and Delphi are more or less on the precision level by Visual C++ from the table top. And if you can want, I can check it truely with test programs with FreePascal and Delphi.

11. **Benjamin Rosseaux**

    August 10, 2010 at 12:51 pm

    So you can download the ObjectPascal test program now from http://vserver.rosseaux.net/stuff/dyadic.zip (with full source code of the test program and my dtoa.c port pasDTOA.pas and two precompiled binaries, compiled with the two available ObjectPascal compilers, one compiled with FreePascal and one compiled with Delphi) to update your table top.

12. **Rick Regan**

    February 25, 2011 at 10:43 am

    2/25/11: I updated the table to reflect the fix for PHP bug 47168.

13. **Brian Gladman**

    January 6, 2012 at 4:22 am

    Hi Rick,

    I just tried your VC++ code on the Visual Studio 11 development preview and got:

    ```
    0.000000059604644775390625
    0.000000029802322387695313
    ```

0.999992370605468750
0.9999961853027343800

So much for Microsoft's commitment to look at this in the next version of the product! It might be worth a complaint!

But, sadly, this matches my experience in reporting numeric weaknesses in VC++ to Microsoft – they just aren't interested in ensuring that Windows is a good basis for scientific or engineering use.

Your site is great and, despite my earlier comment, I do take a look every so often.

Happy New Year to you and your readers

Brian

PS How easy is it for you to maintain your site with WordPress?

---

14. **Rick Regan**

January 6, 2012 at 8:32 am

Hi Brian,

Glad to see you're still around! Thanks for letting us know your results.

I guess "we will look into this for the release after VS2010" is code for "Closed as Won't Fix" 🙂 .

Regarding WordPress, it's easy enough. You do have to stay on top of things and upgrade it (and plugins) periodically though. But I believe the flexibility is worth it.

---

15. **Bruce Dawson**

March 3, 2012 at 12:16 pm

When generating your test numbers why don't you use pow()? That would seem simpler and less error prone than sixteen lines of multiplying 64 digit numbers.

For instance, pow(2.0, -1022 – 52) generates the smallest double.

Having suggested this I need to point out one problem with it. With VC++ if you go pow(2.0, -1022 – 52) you get zero, because it calculates it as 1.0f/pow(2.0, 1022 + 52) and pow(2.0, 1022 + 52) overflows to infinity. However you can still safely calculate it as:

pow(2.0, -1022) * pow(2.0, – 52)

That seems enough simpler to be worth doing.

BTW, I'm doing a floating-point series that overlaps with some of your topics here: http://randomascii.wordpress.com/category/floating-point/

16. **Rick Regan**

March 3, 2012 at 6:37 pm

Hi Bruce,

I was just doing it the easiest, language independent way, one that left me reasonably sure I was getting the correct values. As far as being error prone, I don't think I'll ever need to type them again — I don't see a use outside of this test case (but if I ever need them again, I will consider the power function).

To my readers: Check out Bruce's series on floating-point; it's very enlightening.

17. **Bruce Dawson**

March 12, 2012 at 3:45 am

That's a fair point about wanting your code to be language independent and portable. I tend to value shorter code, but it's not the only metric.

18. Pingback: Float Precision–From Zero to 100+ Digits | Random ASCII

## Comments are closed.

Copyright © 2008-2020 Exploring Binary

Privacy policy

Powered by WordPress