

Hi Francisco,

I am writing in reference to your Service Request # 1-32SLWZ regarding “integer vs. floating point.”

In your e-mail you have asked the following:

- (1) How can `p1(408855776,708158977)` and `p2(408855776,708158977)` be evaluated using integer arithmetics?

Evaluating them using integer arithmetics means that the binary representation of a number differs depending on whether it is defined as an integer or a double. Also the maximum number that can be represented using an integer variable differs. To do integer arithmetic, define the variables as integers as shown below:

```
a1 = int32(408855776); b1 = int32(708158977);
```

The maximum value that can be represented in `int32` type is  $2^{(32-1)}-1$  which is 2147483647. Given the values of `a1` and `b1`, both `a1*a1` as well as `b1*b1` exceed the maximum limit and hence

```
a1*a1 = b1*b1 = 2147483647
```

Hence, for `p1` which is  $2*y^2+9*x^4-y^4$

```
2*y^2+9*x^4 = 2147483647
```

(since each term has already exceeded the maximum limit) and

```
y^4 = 2147483647
```

Hence

```
2*y^2+9*x^4-y^4 = 0
```

The same reasoning can be applied to `p2` to understand how the integer arithmetic is carried on the values.

- (2) Why does R2006a produce `p1(408855776,708158977) = 0`?

In MATLAB, the default data type of a number is double. Hence when we define

```
a = 408855776; b = 708158977;
```

Both `a` and `b` are defined as doubles i.e., the binary representation of the double-precision (or double) data type is defined according to IEEE Standard 754 for double precision. In `p1`, the  $2*y^2$  term is irrelevant because at about  $1e18$  it

is less than half of  $\text{eps}(9*x4) = 3.7e19$  (which is the positive distance from  $\text{abs}(X)$  to the next larger in magnitude floating point number of the same precision as  $X$ ). Thus  $2*y2 + 9*x4 == 9*x4$  in standard double precision arithmetic. The subsequent difference  $9*x4 - y4$  should be zero. Why? Because in exact arithmetic  $9*x4 - y4 = \text{delta}$ , where  $\text{abs}(\text{delta})$  is about  $1e18$ , and  $\backslash\text{eps}(y4) == \text{eps}(9*x4) == 3.7e19$ . Since  $1e18$  is less than half of  $3.7e19$ , the subtraction results in complete loss of precision. All the bits in  $9*x4$  should have been canceled.

However, the result may or may not underflow to zero depending on what the compiler does with any remaining digits in the floating point registers. The result sometimes seems to be re-normalized and returned, even when the compiler is set to use a floating point model which supposedly nullifies any extra precision in the registers.

If you run the following program:

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

double foo(double x,double y) {
    double r1 = x;
    double r2 = y;
    r1 *= r1;
    r1 *= r1;
    r1 *= 9.0;
    r2 *= r2;
    r2 *= r2;
    r1 -= r2;
    return r1;
}

int main() {
    double x = 408855776;
    double y = 708158977;
    double x2,y2;
    x2 = x*x;
    y2 = y*y;
    printf("          9.0*x2*x2 - y2*y2 = %24.16lg\n",9.0*x2*x2 -
y2*y2);
    x2 = x2*x2;
    y2 = y2*y2;
    printf("          9.0*x2 - y2 = %24.16lg\n",9.0*x2 - y2);
    printf("9.0*((x*x)*x)*x - ((y*y)*y)*y = %24.16lg\n",9.0*((x*x)*x)*x -
((y*y)*y)*y);
    printf("          9.0*pow(x,4) - pow(y,4) = %24.16lg\n",9.0*pow(x,4) -
pow(y,4));
}
```

```

printf("                foo(x,y) = %24.16lg\n",foo(x,y));
return 0;
}

```

both GCC and Visual Studio 2005 give

```

9.0*x2*x2 - y2*y2 = 0
9.0*x2 - y2 = -3.68934881474191e+019
9.0*((x*x)*x)*x -((y*y)*y)*y = 0
9.0*pow(x,4) - pow(y,4)=-3.68934881474191e+019
foo(x,y)=-3.68934881474191e+019

```

There is no winning answer here. As I said before, without the leading term of  $p_1$ , the result here should have been about  $-1e18$  in exact arithmetic, and what we see is either zero or  $-\text{eps}(y^4)$ . However, you will be able to get the first one to change to  $-\text{eps}(y^4)$  in Linux by using the `-ffast-math` flag with GCC.

Sometimes it is hard to avoid the little optimizations that compilers do. One way is to do the calculation one step at a time, e.g. in MATLAB or in a spreadsheet like EXCEL. In both cases you get zero, in accordance with the analysis I provided at top, so unless I am missing something, I think the author of this exercise (cited as U. Kulisch.) was trying to illustrate numerical instability and how the values change depending on the internal representation and not really saying that the result has to be the value mentioned in the example for standard integer and floating point representations. As we discussed above the values depend on the little optimizations that the compilers do and are bound to be different depending on those options.

If you need further assistance regarding this issue, please reply to this email preserving the THREAD ID listed below. If you have a new technical support question, please submit a new request here:

[http://www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html)

Sincerely, Jag

Jagadish Gattu  
Application Support Engineer Technical Support  
Department The MathWorks, Inc.  
Phone: (508) 647-7000 option 5  
Email: [support@mathworks.com](mailto:support@mathworks.com)  
Self-Service: <http://www.mathworks.com/support>  
File Exchange and Newsgroup Access: <http://www.mathworks.com/matlabcentral>

[THREAD ID: 1-32SLWZ]