Subject: Re: rounding errors
Posted by Alex Schuster on Fri, 27 Apr 2001 10:52:08 GMT
View Forum Message <> Reply to Message

"Dominic R. Scales" wrote:

> What gives? Is there any numerical math guy/gal out there
> who can tell me how this happens? It seems to me, that
> the accuracy of the second/third cast ist WAY off.
>
> a=double('2.56989')
> b=double(2.56989)

You just have to define the 2.56989 as a double: b = 2.56989d :)

Remember that the floating point value 2.56989 is not the same as the
real number 2.56989000000000000..., but just some value around
2.5698900. Converting it to a double introduces more digits to the
right, but they are pretty random.

Alex
--
Alex Schuster Wonko@weird.cologne.de PGP Key available
alex@pet.mpin-koeln.mpg.de

Subject: Re: rounding errors
Posted by Dominic R. Scales on Fri, 27 Apr 2001 11:38:39 GMT
View Forum Message <> Reply to Message

Alex Schuster wrote:
> You just have to define the 2.56989 as a double: b = 2.56989d :)
>
> Remember that the floating point value 2.56989 is not the same as the
> real number 2.56989000000000000..., but just some value around
> 2.5698900. Converting it to a double introduces more digits to the
> right, but they are pretty random.
>
> Alex
> --
> Alex Schuster Wonko@weird.cologne.de PGP Key available
> alex@pet.mpin-koeln.mpg.de

Danke Alex :)

the prints were more of an example. In the 'real' world,
the data is read in from a file with readu and is an array
of float variables. I want to perform the following maths

in double but with the cast to double I already introduce
'pretty random digits to the right', as you say.
I'ld really like to avoid calling something like double(string(a))
for some large array...

Any ideas?

Schï¿½nes langes Wochenende,
Dominic

[Tuesday is a holiday, so I'll take Monday off :)]
--
Dipl. Phys. Dominic R. Scales | Aero-Sensing Radarsysteme GmbH
Tel: +49 (0)8153-90 88 90 | c/o DLR Oberpfaffenhofen
Fax: +49 (0)8153-908 700 | 82234 Wessling, Germany
WWW: aerosensing.de | email: Dominic.Scales@aerosensing.de

Subject: Re: rounding errors
Posted by Randall Skelton on Fri, 27 Apr 2001 13:17:28 GMT
View Forum Message <> Reply to Message

On Fri, 27 Apr 2001, Dominic R. Scales wrote:

> Danke Alex :)
> the data is read in from a file with readu and is an array
> of float variables. I want to perform the following maths
> in double but with the cast to double I already introduce
> 'pretty random digits to the right', as you say.
> I'ld really like to avoid calling something like double(string(a))
> for some large array...

Perhaps I am confused, but if you want the data to be represented as doubles, you should read it directly into a double array ('array=dblarr(1000)' or something similar)

While you can legitimately extend the precision of floating point numbers to those of doubles you must always remember the underlying IEEE definition which states floats only have 6 digits precision while doubles have 15 digits of precision. When you recast a float into a double, you expect decimal digits 6-15 will be noise because bits beyond the float precision can truly be anything. Asking IDL to make a floating point number into double precision with 'zero padding' like you suggest is like asking IDL to know what decimal digits 6-15 were before you cast them as floats. Using strings as an intermediate type does avoid the problem you describe but it also shows a genuine misunderstanding of storage types.

For the record, I had no idea that IDL requires you to explicitly state 'a=2.348339d0' instead of a=double(2.348339).

Randall

PS: If you are still having trouble with this consider a simple C program:

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

main () {
   float a = 2.38492;  /* original float */
   double b = a;       /* recast */

   double c = 2.38492; /* original double */

   printf("a (float)  = %2.18f\n", a);
   printf("b (recast) = %2.18f\n", b);
   printf("c (double) = %2.18f\n", c);

   return(0);
}
---

[anova ~]% gcc test.c -o test
[anova ~]% ./test
a (float)  = 2.384919881820678711
b (recast) = 2.384919881820678711
c (double) = 2.384920000000000151
```

Subject: Re: rounding errors
Posted by Craig Markwardt on Fri, 27 Apr 2001 13:44:10 GMT
View Forum Message <> Reply to Message

"Dominic R. Scales" <Dominic.Scales@aerosensing.de> writes:

> Alex Schuster wrote:
>> You just have to define the 2.56989 as a double: b = 2.56989d :)
>>
>> Remember that the floating point value 2.56989 is not the same as the
>> real number 2.56989000000000000..., but just some value around
>> 2.5698900. Converting it to a double introduces more digits to the
>> right, but they are pretty random.
>>
>>   Alex
>> --
>>   Alex Schuster   Wonko@weird.cologne.de   PGP Key available

>> alex@pet.mpin-koeln.mpg.de
>
> Danke Alex :)
>
> the prints were more of an example. In the 'real' world,
> the data is read in from a file with readu and is an array
> of float variables. I want to perform the following maths
> in double but with the cast to double I already introduce
> 'pretty random digits to the right', as you say.
> I'ld really like to avoid calling something like double(string(a))
> for some large array...

In principle you should not have to worry about those extra random digits. Since your original data are floating point, then they contain no more than 7 digits of precision. Any digits beyond the 7th are simply *undefined*. If you want higher precision then you should recreate your file using double precision arithmetic from the start.

Even if you use double precision then you will still have random digits at the end, but just farther out:

```
IDL> print, 2.56989d, format='(D30.20)'
       2.56989000000000000767
```

This is a consequence of how numbers are represented in the base-two number system. Generally speaking the relative uncertainty will be (MACHAR()).EPS for floating point in IDL.

Craig

--
-- --------------
Craig B. Markwardt, Ph.D. EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
-- --------------

Subject: Re: rounding errors
Posted by Liam E. Gumley on Fri, 27 Apr 2001 14:21:18 GMT
View Forum Message <> Reply to Message

Randall Skelton wrote:
> For the record, I had no idea that IDL requires you to explicitly state
> 'a=2.348339d0' instead of a=double(2.348339).

This is a subtle but important point. DOUBLE() is a type conversion function, and

a = double(2.348339)

shows a FLOAT argument being converted to a DOUBLE. The safest way to
'cast' a double variable is

a = 2.348339d

or use an array creation function such as DBLARR, DINDGEN, REPLICATE, or
MAKE_ARRAY, e.g.,

b = dblarr(10)
c = dindgen(10)
d = replicate(1.0d, 10)
e = make_array(10, /double)

Cheers,
Liam.
http://cimss.ssec.wisc.edu/~gumley

Subject: Re: rounding errors
Posted by Dominic R. Scales on Fri, 27 Apr 2001 14:41:58 GMT
View Forum Message <> Reply to Message

Craig Markwardt wrote:
>
> In principle you should not have to worry about those extra random
> digits. Since your original data are floating point, then they
> contain no more than 7 digits of precision. Any digits beyond the 7th
> are simply *undefined*. If you want higher precision then you should
> recreate your file using double precision arithmetic from the start.
>
> Even if you use double precision then you will still have random
> digits at the end, but just farther out:
>
> IDL> print, 2.56989d, format='(D30.20)'
> 2.56989000000000000767
>
> This is a consequence of how numbers are represented in the base-two
> number system. Generally speaking the relative uncertainty will be
> (MACHAR()).EPS for floating point in IDL.
>
> Craig

Randall Skelton wrote:
>
> Perhaps I am confused, but if you want the data to be represented as
> doubles, you should read it directly into a double array

> ('array=dblarr(1000)' or something similar)
>
> While you can legitimately extend the precision of floating point numbers
> to those of doubles you must always remember the underlying IEEE
> definition which states floats only have 6 digits precision while doubles
> have 15 digits of precision. When you recast a float into a double, you
> expect decimal digits 6-15 will be noise because bits beyond the float
> precision can truly be anything. Asking IDL to make a floating point
> number into double precision with 'zero padding' like you suggest is like
> asking IDL to know what decimal digits 6-15 were before you cast them as
> floats. Using strings as an intermediate type does avoid the problem
> you describe but it also shows a genuine misunderstanding of storage
> types.
>
> For the record, I had no idea that IDL requires you to explicitly state
> 'a=2.348339d0' instead of a=double(2.348339).
>
> Randall
>
> PS: If you are still having trouble with this consider a simple C program:
>

Dear Craig, dear Randall
thanks for your answers.

Unfortunately, the input data is a fixed format I get from an external
source and can not change. I fully appreciate and understand the precision
difference in significant digits between float and double, be it in idl,
c, or any other programming language, as it is inherently diffucult
(read impossible) to map an infinite set of numbers to a finite realm of
4/8 byte and not miss any.
The case I am trying to make is this: why aren't the missing digits, i.e.
the ones added by the cast, set to 0.? As I go along with my calculations,
these random additional digits give me a hell of a headache by accumulating.
And, as Murphy leads us to expect, always in the 'wrong' direction.
I know, one shouln't test floating point numbers in digital rep. against
one another, not even if the were double.

Starting with numbers that are said to have a precision of 15 digits but
have random digits after after the 6th!?! Why should I even bother?
BUT: it does work for a cast via double(string))...

Yep, it isn't idl's fault here. It only cost me some time to notice...

well, cu all,
Dominic
--
Dipl. Phys. Dominic R. Scales | Aero-Sensing Radarsysteme GmbH

Tel: +49 (0)8153-90 88 90 | c/o DLR Oberpfaffenhofen
Fax: +49 (0)8153-908 700 | 82234 Wessling, Germany
WWW: aerosensing.de | email: Dominic.Scales@aerosensing.de

Subject: Re: rounding errors
Posted by Randall Skelton on Fri, 27 Apr 2001 14:51:05 GMT
View Forum Message <> Reply to Message

On Fri, 27 Apr 2001, Liam E. Gumley wrote:

> This is a subtle but important point. DOUBLE() is a type conversion
> function, and
>
> a = double(2.348339)
>
> shows a FLOAT argument being converted to a DOUBLE. The safest way to
> 'cast' a double variable is
>
> a = 2.348339d
[snip]

Wow... I am glad that I have now learned that particular 'IDL feature'
early on in my PhD. Just yesterday, I convinced the department that we
really need a few good IDL programming books as the current
'learning-by-fire' approach could have some unfortunate consequences ;)

Oh well, back to searching my IDL code for incorrect double precision
constants...

Subject: Re: rounding errors
Posted by thompson on Fri, 27 Apr 2001 15:23:49 GMT
View Forum Message <> Reply to Message

"Dominic R. Scales" <Dominic.Scales@aerosensing.de> writes:

> HELP!

> What gives? Is there any numerical math guy/gal out there
> who can tell me how this happens? It seems to me, that
> the accuracy of the second/third cast ist WAY off.

> a=double('2.56989')
> b=double(2.56989)
> c=double(float('2.56989'))

> print,a,b,c,format='(d)'

> 2.5698900000000000 <---- this is what i want to have
> 2.5698900222778320
> 2.5698900222778320

The first question that comes into my mind is why don't you simply cast your literal as double precision in the first place:

```
IDL> a = 2.56989d0
IDL> print,a,format='(d)'
      2.5698900000000000
```

It's instructive to look at these numbers in the binary representation actually used by the computer. The floating point number 2.56989 has the following representations in hexadecimal and binary formats:

```
Hex:    40247914
Binary: 01000000001001000111100100010100
              ^^^^^^^^^^^^^^^^^^^^^^
```

when this is converted from floating point to double precision, as in your examples b and c above, you get

```
Hex:    40048F2280000000
Binary:  0100000000001001000111100100010100000000000000000000000000000000 0000
               ^^^^^^^^^^^^^^^^^^^^^^
```

It's a little harder to tell in hex format, but the binary format makes it plain that the mantissa part is exactly the same as in the original floating point number, just shifted over a little (the exponent part is bigger in double precision), and filled with zeroes, just as you would expect. The confusion results from the distinction between decimal representation used by people and the binary representation used by computers.

William Thompson

Subject: Re: rounding errors
Posted by John-David T. Smith on Fri, 27 Apr 2001 15:39:21 GMT
View Forum Message <> Reply to Message

"Dominic R. Scales" wrote:
> I fully appreciate and understand the precision
> difference in significant digits between float and double, be it in idl,
> c, or any other programming language, as it is inherently diffucult
> (read impossible) to map an infinite set of numbers to a finite realm of
> 4/8 byte and not miss any.

> The case I am trying to make is this: why aren't the missing digits, i.e.
> the ones added by the cast, set to 0.? As I go along with my calculations,
> these random additional digits give me a hell of a headache by accumulating.
> And, as Murphy leads us to expect, always in the 'wrong' direction.
> I know, one shouln't test floating point numbers in digital rep. against
> one another, not even if the were double.

I think this is a case of not fully understanding how floating point numbers are stored and manipulated (it took me several tries to grasp it). When you have all those trailing zeroes, it's simply a case of the formatted printing routines trying to be clever for you. This cleverness in printout does not extend *in any way* to cleverness in calculation. It's actually not terribly intuitive.

Here is a good discussion (though for fortran): http://www.lahey.com/float.htm. Thanks to whomever originally posted that. A very relevant section:

>>>> >> >>>>>>>
Too Many Digits

You may decide to check the previous program example by printing out both D and X in the same format, like this:

```
30  FORMAT (X, 2G25.15)
PRINT 30, X, D
```

In some FORTRAN implementations, both numbers print out the same. You may walk away satisfied, but you are actually being misled by low-order digits in the display of the single-precision number. In Lahey FORTRAN the numbers are printed out
as:

```
    1.66661000000000        1.66661000251770
```

The reason for this is fairly simple: the formatted-I/O conversion routines know that the absolute maximum decimal digits that have any significance when printing a single-precision entity is 9. The rest of the field is filled with the current
"precision-fill" character, which is "0" by default. The precision-fill character can be changed to any ASCII character, e.g., asterisk or blank. Changing the precision-fill character to "*" emphasizes the insignificance of the low-order digits:

```
    1.66661000******        1.66661000251770
```

>>>> >>> >>>>>>>

JD

Subject: Re: rounding errors
Posted by Paul van Delst on Fri, 27 Apr 2001 15:47:02 GMT
View Forum Message <> Reply to Message

Randall Skelton wrote:
>
> On Fri, 27 Apr 2001, Liam E. Gumley wrote:
>
>> This is a subtle but important point. DOUBLE() is a type conversion
>> function, and
>>
>> a = double(2.348339)
>>
>> shows a FLOAT argument being converted to a DOUBLE. The safest way to
>> 'cast' a double variable is
>>
>> a = 2.348339d
> [snip]
>
> Wow... I am glad that I have now learned that particular 'IDL feature'
> early on in my PhD. Just yesterday, I convinced the department that we
> really need a few good IDL programming books as the current
> 'learning-by-fire' approach could have some unfortunate consequences ;)

This "feature" has absolutely *nothing* to do with IDL. The same thing occurs in other languages, e.g. Fortran, C, etc. Floating point numbers, in general, cannot be represented exactly and you have to keep that in mind when writing code - particularly if the 6th or 7th decimal place is important to you. You could just as well ask why, when you try to this:

IDL> print, (1.0e32)^2

you get the IDL "feature" result:

```
      Inf
% Program caused arithmetic error: Floating overflow
```

Computers have absolutely no intelligence at all - they just do what you tell 'em. If you aren't explicit (e.g. declare a float 2.348339 when you *really* mean 2.3483390000000000) there are defaults they fall back on (e.g. assume single precision float rather than double).

> Oh well, back to searching my IDL code for incorrect double precision
> constants...

No, back to searching your IDL code for your implicitly declared single precision constants and changing them to explicit double precision declarations.

paulv

--
Paul van Delst A little learning is a dangerous thing;
CIMSS @ NOAA/NCEP Drink deep, or taste not the Pierian spring;
Ph: (301)763-8000 x7274 There shallow draughts intoxicate the brain,
Fax:(301)763-8545 And drinking largely sobers us again.
paul.vandelst@noaa.gov Alexander Pope.

Subject: Re: rounding errors
Posted by Craig Markwardt on Fri, 27 Apr 2001 15:52:17 GMT
View Forum Message <> Reply to Message

"Dominic R. Scales" <Dominic.Scales@aerosensing.de> writes:
> Unfortunately, the input data is a fixed format I get from an external
> source and can not change. I fully appreciate and understand the precision
> difference in significant digits between float and double, be it in idl,
> c, or any other programming language, as it is inherently diffucult
> (read impossible) to map an infinite set of numbers to a finite realm of
> 4/8 byte and not miss any.
> The case I am trying to make is this: why aren't the missing digits, i.e.
> the ones added by the cast, set to 0.? As I go along with my calculations,

Why aren't these digits rounded to zero? Because 2.5698900222778320 happens to be the "exact" representation of 2.56989 in 32-bit binary floating point. The grid spacing of floating point numbers increments by 3.06e-7 near a value of 2.56. There is no grid point at 2.56989. Any number between 2.5698900222778320 +/- (0.5 * 3.06e-7) will collapse to that number. You can verify this with the following statements:

```
IDL> ff = 2.5698900222778320 + (findgen(100)-50)*3.06e-7/20
IDL> print, ff, format='(2(F24.20))'
```

Notice how the input values all collapse to a just a few output values.

> these random additional digits give me a hell of a headache by accumulating.
> And, as Murphy leads us to expect, always in the 'wrong' direction.
> I know, one shouln't test floating point numbers in digital rep. against
> one another, not even if the were double.

If the errors are accumulating signifcantly in your calculations then you have other problems. Remember, in your original float data the fractional uncertainty is *always* +/- 1.2e-7. You can never get around this. If this is making a difference in your output results then your external source is obviously not providing high enough

precision data.

You can estimate confidence intervals on your output values by
shifting the inputs by +/- 1 bit, and seeing how the results change.

```
eps = (machar()).eps
fcent  = f(data)
fplus  = f(data * (1.+eps) )
fminus = f(data * (1.-eps) )
```

where F is your computational function. If FCENT, FPLUS and FMINUS
are intolerably different then you know that the problem is in the
source of your data, and not in IDL. Of course, there are numerical
techniques that you can apply within your own calculation which may
help avoid cancellation errors. For example, the expression

```
  sqrt(1+x) - 1
```

is woefully inaccurate when X is small. Routines like HYPOT in
Numerical Recipes, which seem on their face to be superfluous, treat
this kind of problem.

Craig

--

 -- --------------
Craig B. Markwardt, Ph.D.   EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
 -- --------------

Subject: Re: rounding errors
Posted by Karl Schultz on Fri, 27 Apr 2001 16:07:21 GMT
View Forum Message <> Reply to Message

You could try

```
d = 2.56989d
```

to use a double-precision literal.

Also,

```
b=double(2.56989)
```

causes a single-to-double conversion because the literal is
single-precision.

The IEEE mantissa (fractional part) of this single-precision literal is (in hex): 48F228
The conversion of a single to double will make the mantissa something like 48F2280000000 because it just adds zeros to pad out the mantissa.
But the mantissa of 2.56989d is 48F227D028A1E.

Comparing:
48F2280000000 (single-precision literal converted to double)
48F227D028A1E (double precision literal stored as double)

The first number is slightly larger, which accounts for the extra non-zero decimal digits you point out below. When you convert the 2.56989 literal to a single-precision float, the last mantissa bit is rounded up to get as close to the literal as possible. When you convert that single to a double, the "too high" bit is simply carried over to the double precision mantissa and the rest of the lesser-significant bits are zeroed out. So, the number still seems "too high". But if you store a double-precision literal as a double, all the bits in a double-precision mantissa are used and the result is what you expect.

And '2.56989' is different from 2.56989 because the latter is an implied single-precision floating point number. The former is a string which gets converted directly to double-precision if you say double('2.56989').

"Dominic R. Scales" <Dominic.Scales@aerosensing.de> wrote in message news:3AE9330C.29D059E9@aerosensing.de...
> HELP!
>
> What gives? Is there any numerical math guy/gal out there
> who can tell me how this happens? It seems to me, that
> the accuracy of the second/third cast ist WAY off.
>
>   a=double('2.56989')
>   b=double(2.56989)
>   c=double(float('2.56989'))
> .
>   print,a,b,c,format='(d)'
>
>   2.5698900000000000 <---- this is what i want to have
>   2.5698900222778320
>   2.5698900222778320
>

Subject: Re: rounding errors
Posted by Liam E. Gumley on Fri, 27 Apr 2001 17:01:15 GMT

JD Smith wrote:
> Here is a good discussion (though for fortran):
> http://www.lahey.com/float.htm.

Another informative discussion of floating point numbers is available in section 4 of

http://www.ibiblio.org/pub/languages/fortran/unfp.html

Cheers,
Liam.

Subject: Re: rounding errors
Posted by thompson on Fri, 27 Apr 2001 18:34:07 GMT

"Dominic R. Scales" <Dominic.Scales@aerosensing.de> writes:

> Unfortunately, the input data is a fixed format I get from an external
> source and can not change. ...

When you say that it's in a fixed format, do you mean an ASCII format (READF), or a binary format (READU)? If the former, then you can do exactly what you want by reading it in as binary in the first place, e.g.

```
A = 0.D0
READF, UNIT, A
```

for a single value, or

```
A = DBLARR(N_VALUES)
READF, UNIT, A
```

for an array. You will at least then retain the precision of all the digits that were printed out.

Digits not printed out, on the other hand, are lost forever. If your file contains the number 2.56989 printed out as a series of digits, it's rather naive to think of this as 2.569890000000..., it really could have been anything from 2.569885 to 2.569894999... before it was rounded off to six digits of precision.

If you're instead reading in single precision binary numbers via READU, then you've already made one approximation when you've printed it out as 2.56989. Binary data aren't stored in decimal notation.

> The case I am trying to make is this: why aren't the missing digits, i.e.
> the ones added by the cast, set to 0.? ...

They are set to 0, in binary notation, as I demonstrated in my last post.
They're not really "random". It's just the conversion from binary to decimal
notation that makes it look so.

> ... As I go along with my calculations,
> these random additional digits give me a hell of a headache by accumulating.

But why would you expect that treating 2.56989 as 2.5698900000000000 is any
more accurate than 2.5698900222778320? It's only a human prejudice to want to
write it out that way. Computers have a different prejudice to fill out with
zeros in binary notation rather than decimal, but that's just as valid as the
human prejudice towards decimal notation. In fact, this particular result is
two orders of magnitude closer to what you want than you have any right to
expect, differing only in the 9th digit.

> And, as Murphy leads us to expect, always in the 'wrong' direction.
> I know, one shouln't test floating point numbers in digital rep. against
> one another, not even if the were double.

> Starting with numbers that are said to have a precision of 15 digits but
> have random digits after after the 6th!?! Why should I even bother?
> BUT: it does work for a cast via double(string))...

But if your data is only stored with 6 digit accuracy, how can you hope to
recover 15 digit accuracy. Filling the data out with zeros in decimal (or
binary) notation is really making up data out of thin air. It's not really
justified, except that one has to fill it up with something.

> Yep, it isn't idl's fault here. It only cost me some time to notice...

> well, cu all,
> Dominic
> --
> Dipl. Phys. Dominic R. Scales | Aero-Sensing Radarsysteme GmbH
> Tel: +49 (0)8153-90 88 90 | c/o DLR Oberpfaffenhofen
> Fax: +49 (0)8153-908 700 | 82234 Wessling, Germany
> WWW: aerosensing.de | email: Dominic.Scales@aerosensing.de

Subject: Re: rounding errors
Posted by James Kuyper on Wed, 02 May 2001 15:06:19 GMT
View Forum Message <> Reply to Message

"Dominic R. Scales" wrote:

>
> Craig Markwardt wrote:
>>
>> In principle you should not have to worry about those extra random
>> digits. Since your original data are floating point, then they
>> contain no more than 7 digits of precision. Any digits beyond the 7th
>> are simply *undefined*. If you want higher precision then you should
>> recreate your file using double precision arithmetic from the start.
>>
>> Even if you use double precision then you will still have random
>> digits at the end, but just farther out:
>>
>> IDL> print, 2.56989d, format='(D30.20)'
>> 2.56989000000000000767
>>
>> This is a consequence of how numbers are represented in the base-two
>> number system. Generally speaking the relative uncertainty will be
>> (MACHAR()).EPS for floating point in IDL.
>>
>> Craig
>
> Randall Skelton wrote:
>>
>> Perhaps I am confused, but if you want the data to be represented as
>> doubles, you should read it directly into a double array
>> ('array=dblarr(1000)' or something similar)
>>
>> While you can legitimately extend the precision of floating point numbers
>> to those of doubles you must always remember the underlying IEEE
>> definition which states floats only have 6 digits precision while doubles
>> have 15 digits of precision. When you recast a float into a double, you
>> expect decimal digits 6-15 will be noise because bits beyond the float
>> precision can truly be anything. Asking IDL to make a floating point
>> number into double precision with 'zero padding' like you suggest is like
>> asking IDL to know what decimal digits 6-15 were before you cast them as
>> floats. Using strings as an intermediate type does avoid the problem
>> you describe but it also shows a genuine misunderstanding of storage
>> types.
>>
>> For the record, I had no idea that IDL requires you to explicitly state
>> 'a=2.348339d0' instead of a=double(2.348339).
>>
>> Randall
>>
>> PS: If you are still having trouble with this consider a simple C program:
>>
>
> Dear Craig, dear Randall

> thanks for your answers.
>
> Unfortunately, the input data is a fixed format I get from an external
> source and can not change. I fully appreciate and understand the precision
> difference in significant digits between float and double, be it in idl,
> c, or any other programming language, as it is inherently diffucult
> (read impossible) to map an infinite set of numbers to a finite realm of
> 4/8 byte and not miss any.
> The case I am trying to make is this: why aren't the missing digits, i.e.
> the ones added by the cast, set to 0.? As I go along with my calculations,

There are no "extra" digits being set to 0, because there aren't any digits, extra or otherwise. Every single bit in the mantissa is being used to represent the number you requested as accurately as possible as a binary fraction. It simply can't be represented as an exact binary fraction. This isn't unusual; it's true for most numbers that can be represented by a decimal fraction. This isn't unique to binary formats; no matter what base you use, most numbers can't be represented exactly in a finite number of digits, even if you restrict yourself to rational numbers.

This is why many early computers experimented with various binary coded decimal formats; basically, each half of a byte was used to store a single decimal digit. Decimal numbers could be represented exactly. However, that advantage was not sufficient to sustain them. These schemes have been pretty much abandoned in favor of pure binary formats. I can't vouch for the reason, but I suspect that the hardware implementation of math operations on BCD formats was woefully inefficient compared to using true binary formats.

Subject: Re: rounding errors
Posted by James Kuyper on Wed, 02 May 2001 15:07:23 GMT
View Forum Message <> Reply to Message

Paul van Delst wrote:
>
> Randall Skelton wrote:
>>
>> On Fri, 27 Apr 2001, Liam E. Gumley wrote:
>>
>>> This is a subtle but important point. DOUBLE() is a type conversion
>>> function, and
>>>
>>> a = double(2.348339)
>>>
>>> shows a FLOAT argument being converted to a DOUBLE. The safest way to
>>> 'cast' a double variable is

>>>
>>> a = 2.348339d
>> [snip]
>>
>> Wow... I am glad that I have now learned that particular 'IDL feature'
>> early on in my PhD. Just yesterday, I convinced the department that we
>> really need a few good IDL programming books as the current
>> 'learning-by-fire' approach could have some unfortunate consequences ;)
>
> This "feature" has absolutely *nothing* to do with IDL. The same thing occurs in other
> languages, e.g. Fortran, C, etc. Floating point numbers, in general, cannot be represented
> exactly and you have to keep that in mind when writing code

I think you're misunderstanding the "feature" of IDL that surprised me
as much as it surprised Randall and Liam. This has everything to do with
IDL, and nothing to do with expecting exact representation of a
finite-length decimal fraction. By default, in C 2.348339 represents a
double precision number, not a single precision one, and I'd never
realized that the IDL convention was different. That probably explains a
number of the wierd results I've seen.
