Testing

Floating Point

Still looking at fields in Saskatchewan...

Still looking at fields in Saskatchewan...

Fields				
Corner 1		Corner 2		Crop
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮

Still looking at fields in Saskatchewan...

Latitude/longitude

Fields				
Corner 1		Corner 2		Crop
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮

Still looking at fields in Saskatchewan...

Latitude/longitude

Floating point numbers

Fields				
Corner 1		Corner 2		Crop
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮

Still looking at fields in Saskatchewan...

Latitude/longitude

Floating point numbers

That's when trouble starts

Fields				
Corner 1		Corner 2		Crop
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮

Finding a good representation for floating-point numbers is hard

Finding a good representation for floating-point numbers is hard

Can't actually represent an infinite number of real values with a finite set of bit patterns

Finding a good representation for floating-point numbers is hard

Can't actually represent an infinite number of real values with a finite set of bit patterns

What follows is (over-)simplified

Finding a good representation for floating-point numbers is hard

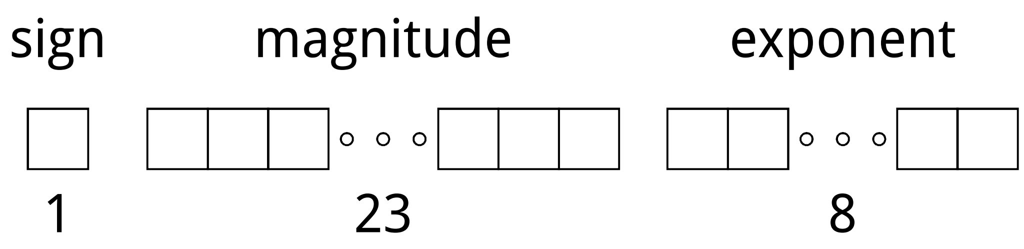
Can't actually represent an infinite number of real values with a finite set of bit patterns

What follows is (over-)simplified

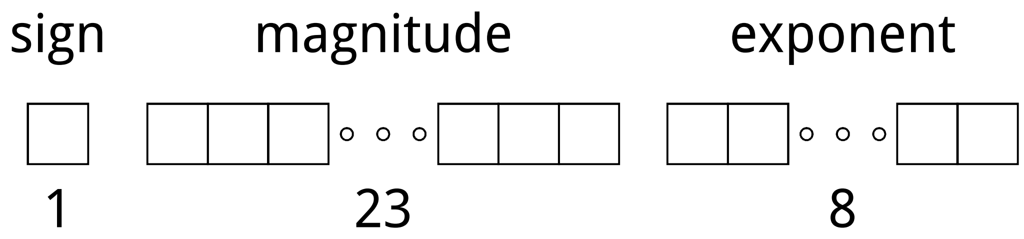
Goldberg (1991): "What Every Computer Scientist Should Know About Floating-Point Arithmetic"

Use sign, magnitude, and exponent

Use sign, magnitude, and exponent

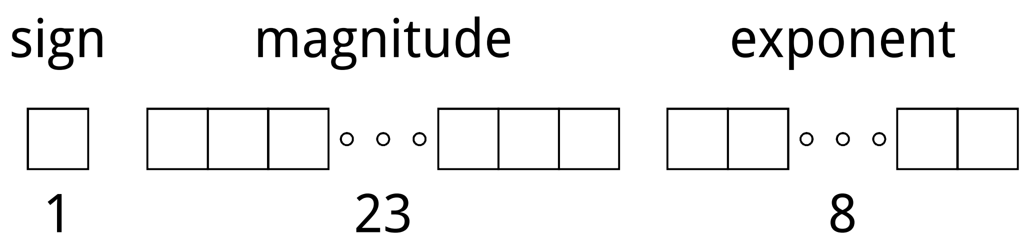


Use sign, magnitude, and exponent



To illustrate problems, we'll use a simpler format

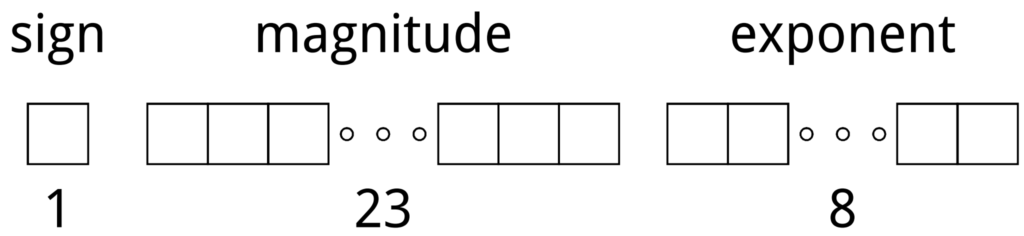
Use sign, magnitude, and exponent



To illustrate problems, we'll use a simpler format

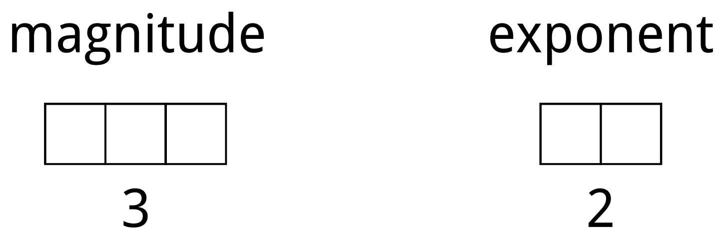
And only positive values without fractions

Use sign, magnitude, and exponent



To illustrate problems, we'll use a simpler format

And only positive values without fractions



Possible values

		Exponent		
Mantissa	00	01	10	11
000	0	0	0	0
001	1	2	4	8
010	2	4	8	16
011	3	6	12	24
100	4	8	16	32
101	5	10	20	40
110	6	12	24	48
101	7	14	28	56

Possible values

Exponent

Mantissa	00	01	10	11
000	0	0	0	0
001	1	2	4	8
010	2	4	8	16
011	3	6	12	24
100	4	8	16	32
101	5	10	20	40
110	6	12	24	48
101	7	14	28	56

$\underline{110} \times 2^{\underline{11}}$

Possible values

Mantissa \ Exponent	00	01	10	11
000	0	0	0	0
001	1	2	4	8
010	2	4	8	16
011	3	6	12	24
100	4	8	16	32
101	5	10	20	40
110	6	12	24	48
101	7	14	28	56

$\underline{110} \times 2^{\underline{11}}$

6×2^3

Possible values

Exponent

Mantissa	00	01	10	11
000	0	0	0	0
001	1	2	4	8
010	2	4	8	16
011	3	6	12	24
100	4	8	16	32
101	5	10	20	40
110	6	12	24	48
101	7	14	28	56

$\underline{110} \times 2^{\underline{11}}$

6×2^3

6×8

Possible values

Exponent

Mantissa	00	01	10	11
000	0	0	0	0
001	1	2	4	8
010	2	4	8	16
011	3	6	12	24
100	4	8	16	32
101	5	10	20	40
110	6	12	24	48
101	7	14	28	56

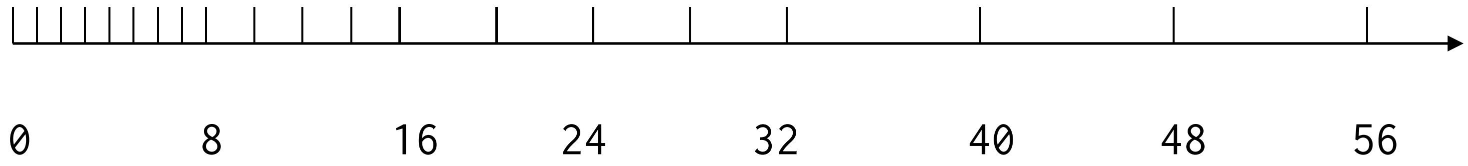
$\underline{110} \times 2^{\underline{11}}$

6×2^3

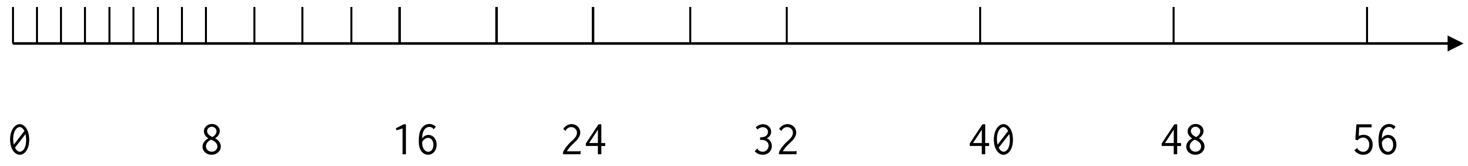
6×8

Actual representation doesn't have redundancy

A clearer view of those values

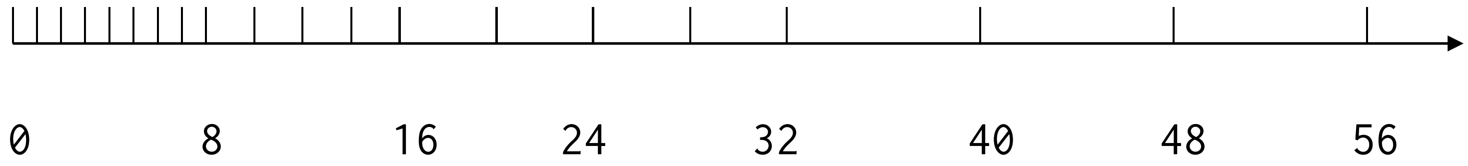


A clearer view of those values



There are numbers we can't represent

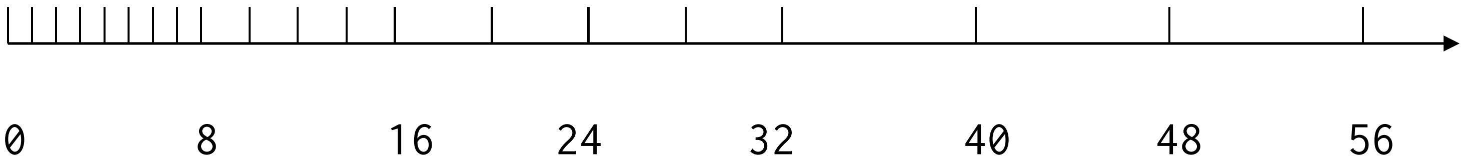
A clearer view of those values



There are numbers we can't represent

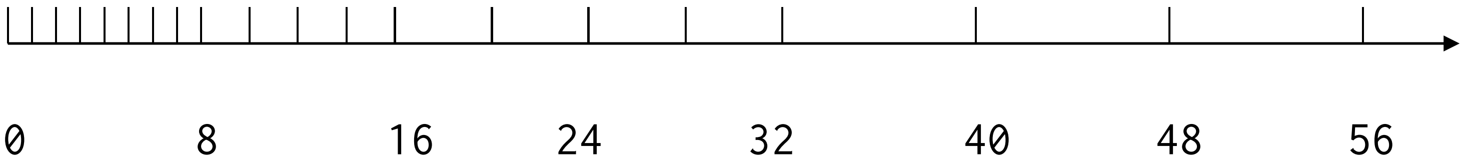
Just as 1/3 must be 0.3333 or 0.3334 in decimal

A clearer view of those values



This scheme has no representation for 9

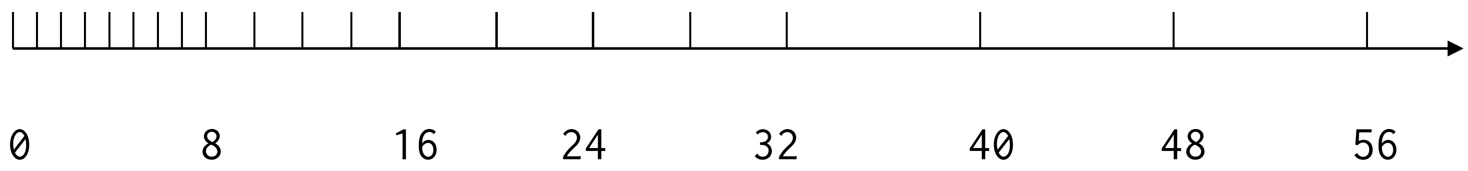
A clearer view of those values



This scheme has no representation for 9

So 8+1 must be either 8 or 10

A clearer view of those values

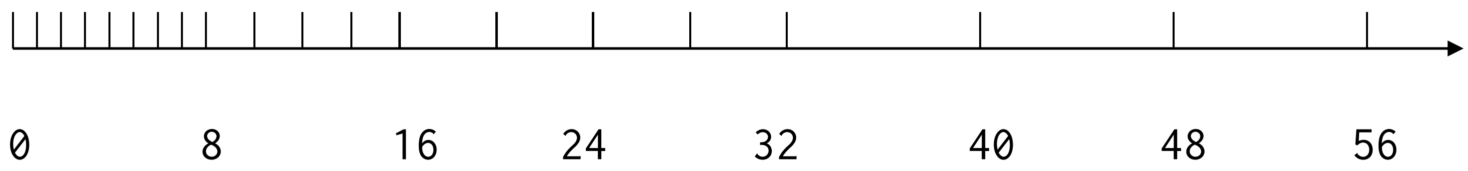


This scheme has no representation for 9

So 8+1 must be either 8 or 10

If 8+1 = 8, what is 8+1+1?

A clearer view of those values



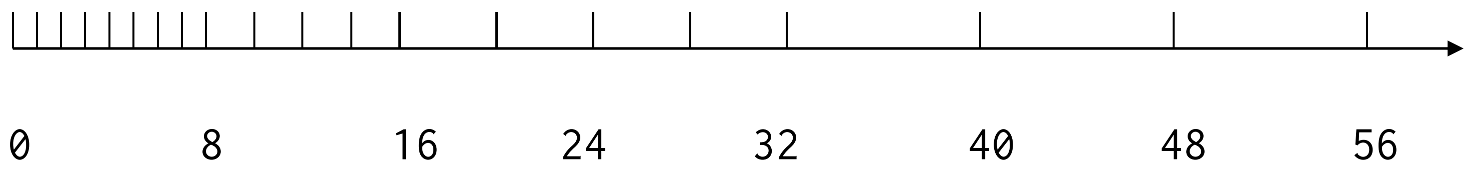
This scheme has no representation for 9

So 8+1 must be either 8 or 10

If 8+1 = 8, what is 8+1+1?

(8+1)+1 = 8+1 (if we round down) = 8 again

A clearer view of those values



This scheme has no representation for 9

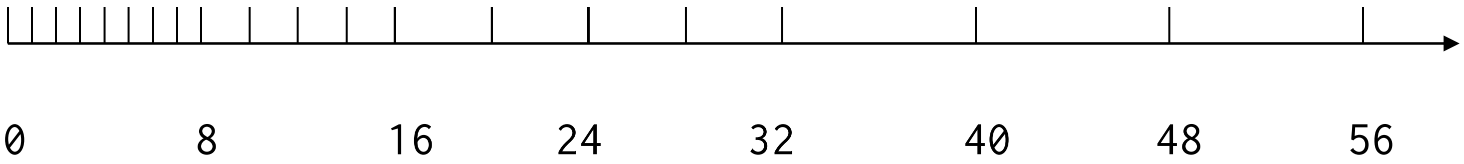
So 8+1 must be either 8 or 10

If 8+1 = 8, what is 8+1+1?

(8+1)+1 = 8+1 (if we round down) = 8 again

But 8+(1+1) = 8+2 = 10, which we *can* represent

A clearer view of those values



This scheme has no representation for 9

So 8+1 must be either 8 or 10

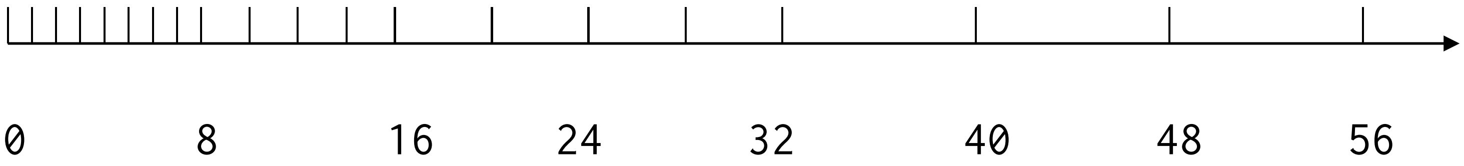
If 8+1 = 8, what is 8+1+1?

(8+1)+1 = 8+1 (if we round down) = 8 again

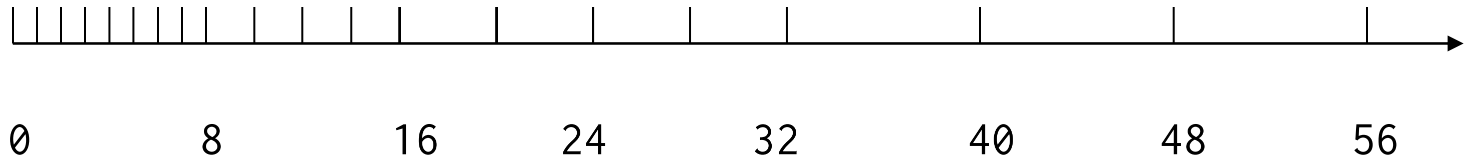
But 8+(1+1) = 8+2 = 10, which we *can* represent

"Sort then sum" would give the same answer...

Another observation

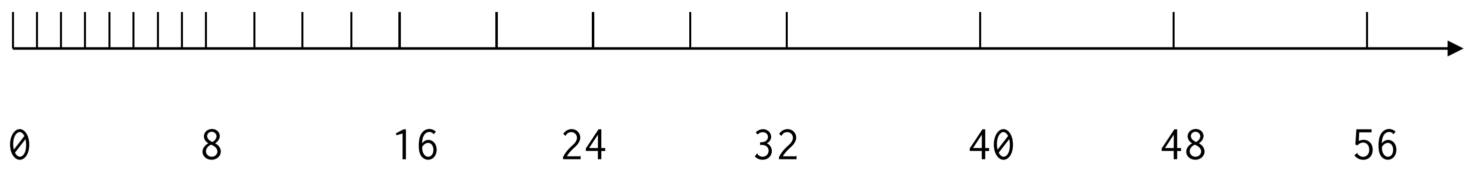


Another observation



Spacing is uneven

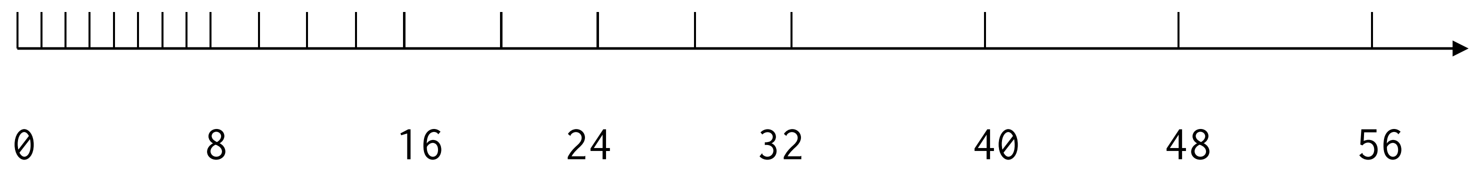
Another observation



Spacing is uneven

But *relative* spacing stays the same

Another observation



Spacing is uneven

But *relative* spacing stays the same

Because we're multiplying the same few mantissas

by ever-larger exponents

Absolute error = |val – approx|

Absolute error = |val – approx|

Relative error = |val – approx| / val

Absolute error = |val – approx|

Relative error = |val – approx| / val

Operation	Actual Result	Intended Result	Absolute Error	Relative Error
8+1	8	9	1	1/9 (11.11%)
56+1	56	57	1	1/57 (1.75%)

Absolute error = |val – approx|

Relative error = |val – approx| / val

Operation	Actual Result	Intended Result	Absolute Error	Relative Error
8+1	8	9	1	1/9 (11.11%)
56+1	56	57	1	1/57 (1.75%)

Relative error is more useful

Absolute error = |val – approx|

Relative error = |val – approx| / val

Operation	Actual Result	Intended Result	Absolute Error	Relative Error
8+1	8	9	1	1/9 (11.11%)
56+1	56	57	1	1/57 (1.75%)

Relative error is more useful

Makes little sense to say "off by 0.01" if the value you're approximating is 0.000000001

What does this have to do with testing?

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

i = 1, 2, 3, ..., 9

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

0.9, 0.09, 0.009, ...

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

0.9, 0.99, 0.999, ...

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
       (i, total, total-expected)
```

But is it?

0.9, 0.99, 0.999, ...

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

1-0.1, 1-0.01, ...

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

Should also make

0.9, 0.99, ...

1-0.1, 1-0.01, ...

What does this have to do with testing?

```
vals = []
for i in range(1, 10):
  number = 9.0 * 10.0 ** -i
  vals.append(number)
  total = sum(vals)
  expected = 1.0 - (1.0 * 10.0 ** i)
  diff = total - expected
  print '%2d  %22.21f  %22.21f' % \
        (i, total, total-expected)
```

Check and print

And the answer is:

```
1  0.90000000000000022204  0.00000000000000000000
2  0.98999999999999991118  0.00000000000000000000
3  0.99899999999999999112  0.00000000000000000000
4  0.99990000000000001101  0.00000000000000000000
5  0.99999000000000004551  0.00000000000000000000
6  0.99999900000000008227  0.00000000000000111022
7  0.99999990000000005264  0.00000000000000000000
8  0.99999999000000006077  0.00000000000000111022
9  0.99999999900000002828  0.00000000000000000000
```

And the answer is:

1	0.9000000000000000022204	0.0000000000000000000000
2	0.9899999999999999991118	0.0000000000000000000000
3	0.9989999999999999999112	0.0000000000000000000000
4	0.99990000000 [illegible]	[illegible] 00000000000000
5	0.999990000000 [illegible]	[illegible] 00000000000000
6	0.9999990000000000082267	0.0000000000000000111022
7	0.9999999000000000052636	0.0000000000000000000000
8	0.9999999900000000060775	0.0000000000000000111022
9	0.9999999990000000028282	0.0000000000000000000000

Already slightly off

And the answer is:

1	0.90000000000000022204	0.00000000000000000000
2	0.98999999999999991118	0.00000000000000000000
3	0.99899999999999999112	0.00000000000000000000
4	0.999	00000
5	0.999	00000
6	0.99999900000000082267	0.00000000000000111022
7	0.99999990000000052636	0.00000000000000000000
8	0.99999999000000060775	0.00000000000000111022
9	0.99999999900000028282	0.00000000000000000000

But at least they're consistent

And the answer is:

1	0.900000000000000022204	0.000000000000000000000
2	0.989999999999999991118	0.000000000000000000000
3	0.998999999999999999112	0.000000000000000000000
4	0.999900000000000011013	0.000000000000000000000
5	0.999990000000000045510	0.000000000000000000000
6	0.999999000000000082267	0.000000000000000111022
7	0.999999900000000052636	0.000000000000000000000
8	0.999999990000000060775	0.000000000000000111022
9	0.999999999000000028282	0.000000000000000000000

Sometimes, they're not

And the answer is:

1	0.9000000000000000022204	0.0000000000000000000000
2	0.9899999999999999991118	0.0000000000000000000000
3	0.9989999999999999999112	0.0000000000000000000000
4	0.9999000000000000011013	0.0000000000000000000000
5	0.9999900000000000045510	0.0000000000000000000000
6	0.9999990000000000082267	0.0000000000000000111022
7	0.9999999000000000052636	0.0000000000000000000000
8	0.9999999900000000060775	0.0000000000000000111022
9	0.9999999990000000028282	0.0000000000000000000000

Sometimes errors cancel out later

So what does this have to do with testing?

So what does this have to do with testing?

What do you compare the actual test result to?

So what does this have to do with testing?

What do you compare the actual test result to?

`sum(vals[0:7]) = 0.9999999` could well be `False`

So what does this have to do with testing?

What do you compare the actual test result to?

`sum(vals[0:7]) = 0.9999999` could well be `False`

Even if `vals = [0.9, 0.09, ...]`

So what does this have to do with testing?

What do you compare the actual test result to?

`sum(vals[0:7]) = 0.9999999` could well be `False`

Even if `vals = [0.9, 0.09, ...]`

And there are no bugs in our code

So what does this have to do with testing?

What do you compare the actual test result to?

`sum(vals[0:7]) = 0.9999999` could well be `False`

Even if `vals = [0.9, 0.09, ...]`

And there are no bugs in our code

This is *hard*

So what does this have to do with testing?

What do you compare the actual test result to?

`sum(vals[0:7]) = 0.9999999` could well be `False`

Even if `vals = [0.9, 0.09, ...]`

And there are no bugs in our code

This is *hard*

No one has a good answer

What can you do?

What can you do?

1. Compare to analytic solutions (when you can)

What can you do?

1. Compare to analytic solutions (when you can)
2. Compare complex to simple

What can you do?

1. Compare to analytic solutions (when you can)
2. Compare complex to simple
3. Never use == or != with floating-point numbers

What can you do?

1. Compare to analytic solutions (when you can)
2. Compare complex to simple
3. Never use == or != with floating-point numbers

   (It's OK to trust <, >=, etc.)

What can you do?

1. Compare to analytic solutions (when you can)
2. Compare complex to simple
3. Never use == or != with floating-point numbers

   (It's OK to trust <, >=, etc.)

Use `nose.assert_almost_equals(expected, actual)`

What can you do?

1. Compare to analytic solutions (when you can)
2. Compare complex to simple
3. Never use == or != with floating-point numbers

   (It's OK to trust <, >=, etc.)

Use `nose.assert_almost_equals(expected, actual)`

```
Fail if the two objects are unequal as determined by
their difference rounded to the given number of
decimal places (default 7) and comparing to zero.
```

What can you do?

1. Compare to analytic solutions (when you can)
2. Compare complex to simple
3. Never use == or != with floating-point numbers

   (It's OK to trust <, >=, etc.)

Use `nose.assert_almost_equals(expected, actual)`

```
Fail if the two objects are unequal as determined by
their difference rounded to the given number of
decimal places (default 7) and comparing to zero.
```

Is that absolute or relative?

created by

Greg Wilson

August 2010