Notes for Chapter 1 of

Scientific Computing with Case Studies

Dianne P. O'Leary
SIAM Press, 2008

- Mathematical modeling
- Computer arithmetic
- Errors

Arithmetic and Error

What we need to know about error:

-- how does error arise

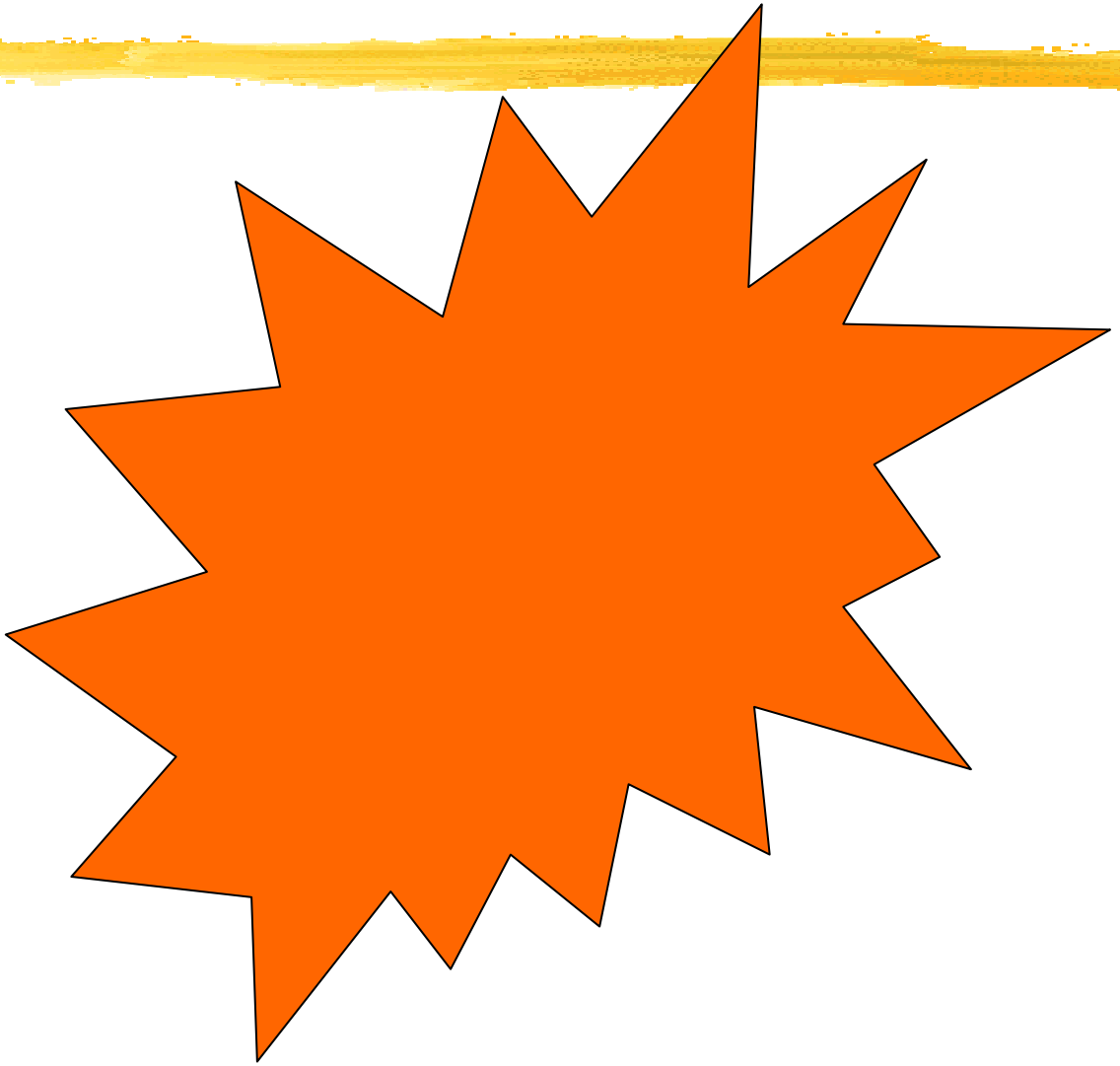
-- how machines do arithmetic

- -- fixed point arithmetic
- -- floating point arithmetic

-- how errors are propagated in calculations.

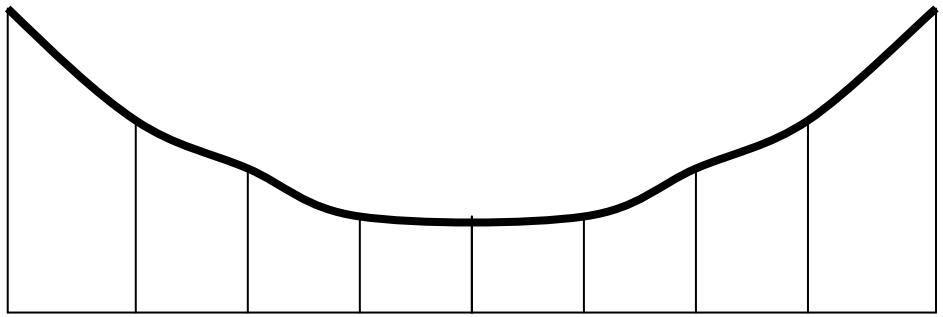
-- how to measure error

How does error arise?



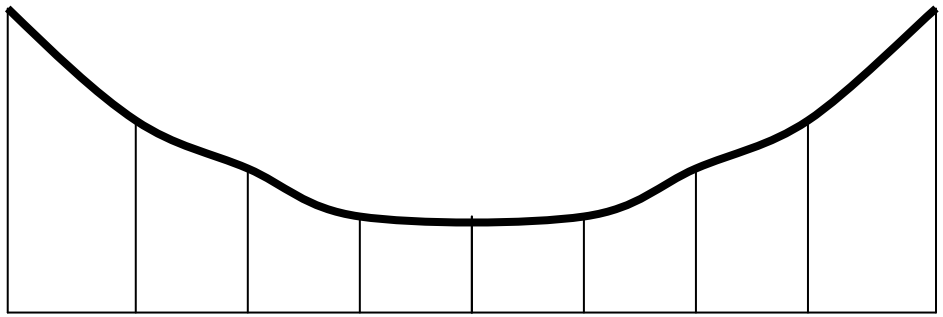
How does error arise?

Example: An engineer wants to study the stresses in a bridge.



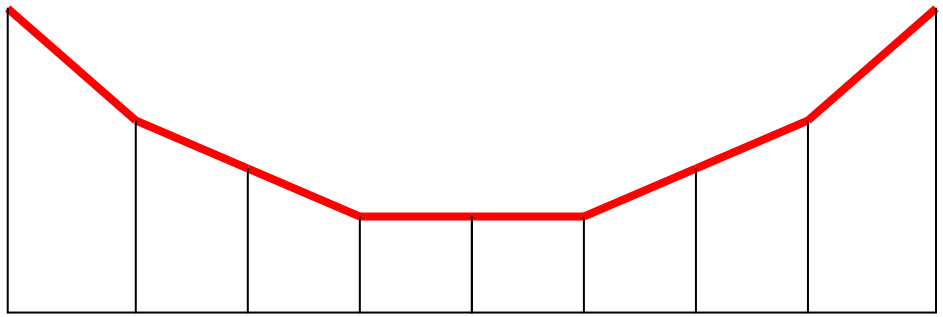
Measurement error

Step 1: Gather lengths, angles, etc. for girders and wires.



Modeling error

Step 2: Approximate system by finite elements.



Truncation error

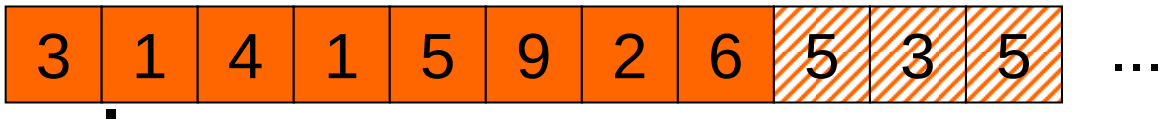
Step 3: Numerical analyst develops an algorithm: the stress can be computed as the **limit** (as n becomes infinite) of some function G(n).

Can't take this limit on a computer, so decide to use G(150).

G(1), G(2), G(3), G(4), G(5), ...

Roundoff error

Step 4: The algorithm is programmed and run on a computer. We need π. Approximate it by 3.1415926.



Sources of error

1. Measurement error
2. Modeling error
3. Truncation error
4. Rounding error

No mistakes!

Note: No **mistakes**:

- the engineer did not misread ruler,
- the model was good,
- the programmer did not make a typo in the definition of π,
- and the computer worked flawlessly.

But the engineer will want to know what the final answer has to do with the stresses on the real bridge!

What does a numerical analyst do?

-- design algorithms and analyze them.

-- develop mathematical software.

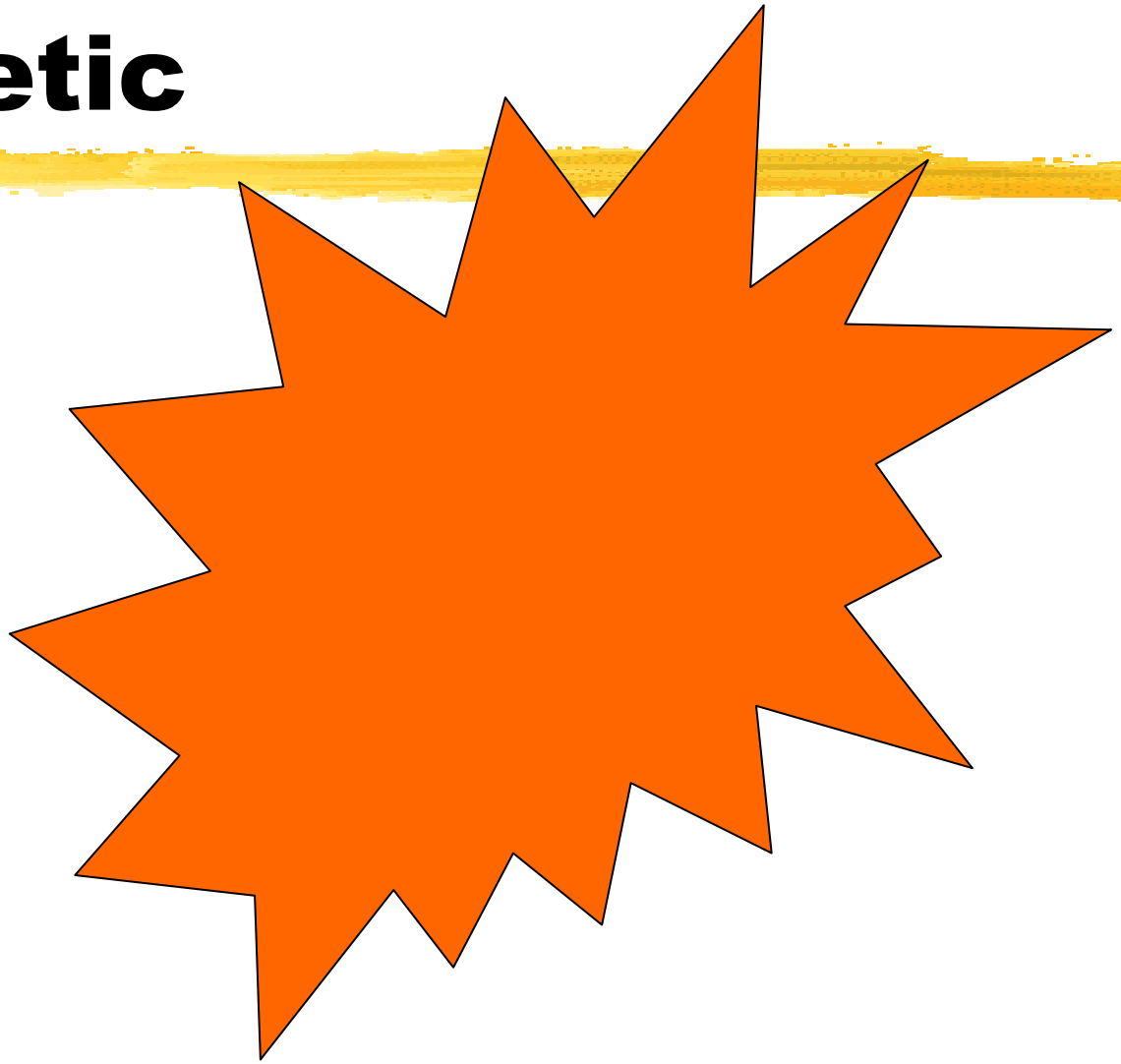
-- answer questions about how accurate the final answer is.

What does a computational scientist do?

-- works as part of an interdisciplinary team.

-- intelligently uses mathematical software to analyze mathematical models.

How machines do arithmetic

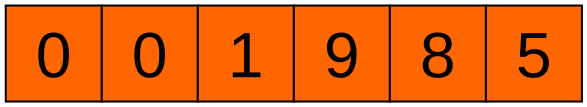


Machine Arithmetic: Fixed Point

How integers are stored in computers:

Each **word** (storage location) in a machine contains a fixed number of digits.

Example: A machine with a 6-digit word might represent 1985 as



Fixed Point: Decimal vs. Binary

Most calculators use **decimal (base 10)** representation.

Each digit is an integer between 0 and 9.

The value of the number is

$$1 \times 10^3 + 9 \times 10^2 + 8 \times 10^1 + 5 \times 10^0 .$$

Fixed Point: Decimal vs. Binary

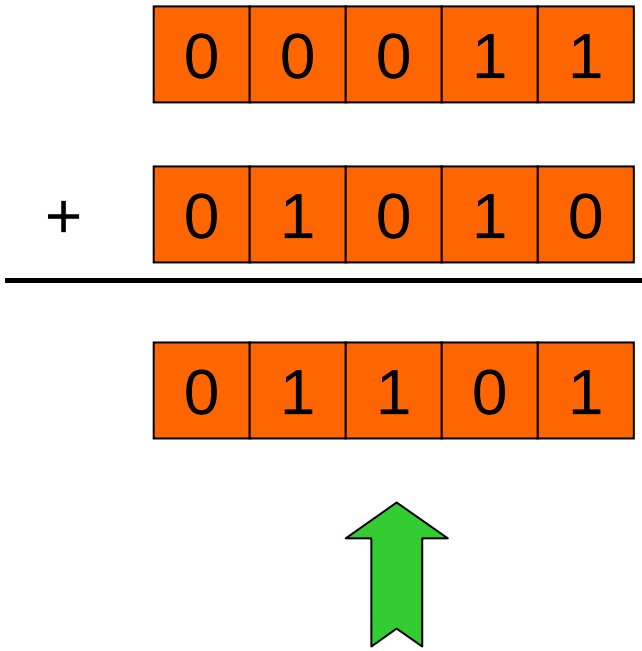
Most computers use **binary (base 2)** representation.

Each digit is the integer 0 or 1.

If the number above is binary, its value is

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 .$$ (or 22 in base 10)

Example of binary addition



0 + 0 = 0

0 + 1 = 1

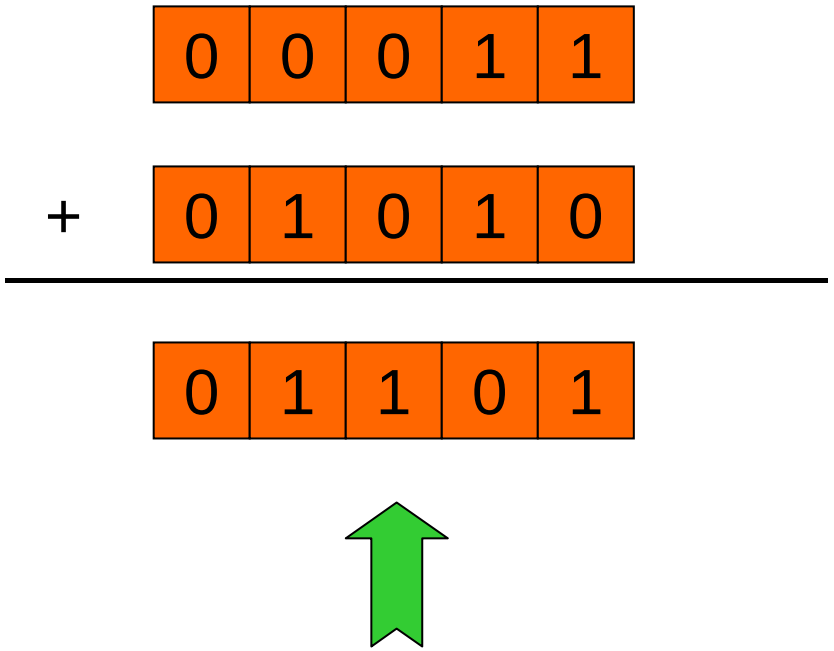
1 + 0 = 1

1 + 1 = 10 (binary) = 10_2 = 2

1 + 1 + 1 = 11 (binary)=11_2 = 3

Note the "**carry**" here!

Example of binary addition



In decimal notation, 3

+10

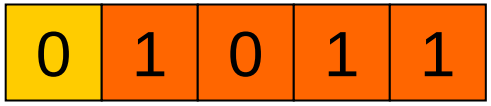
=13

Note the "**carry**" here!

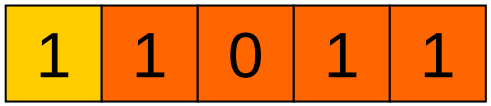
Representing negative numbers

Computers represent negative numbers using "one's complement", "two's complement", or sign-magnitude representation.

Sign magnitude is easiest, and enough for us: if the first bit is zero, then the number is positive. Otherwise, it is negative.

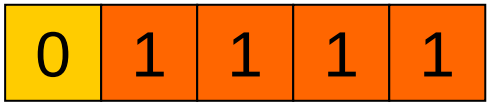


Denotes +11.



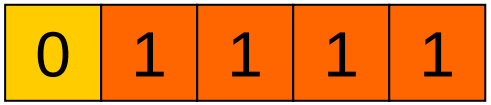
Denotes - 11.

Range of fixed point numbers

Largest 5-digit (5 bit) binary number: 

= 15

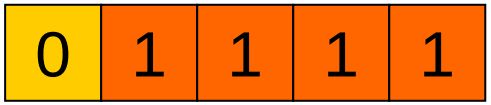
Range of fixed point numbers

Largest 5-digit (5 bit) binary number: 

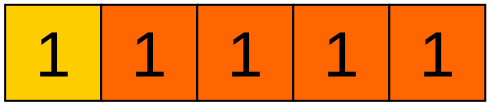
= 15

Smallest:

Range of fixed point numbers

Largest 5-digit (5 bit) binary number: 

= 15

Smallest: 

= -15

Smallest positive:

Range of fixed point numbers

Largest 5-digit (5 bit) binary number: 0 1 1 1 1 = 15

Smallest: 1 1 1 1 1 = -15

Smallest positive: 0 0 0 0 1 = 1

Overflow

If we try to add these numbers:

	0	1	1	1	1	= 15
+	0	1	0	0	0	= 8

we get

1	0	1	1	1	= -7.

We call this overflow: the answer is too large to store, since it is outside the range of this number system.

Features of fixed point arithmetic

Easy: always get an integer answer.

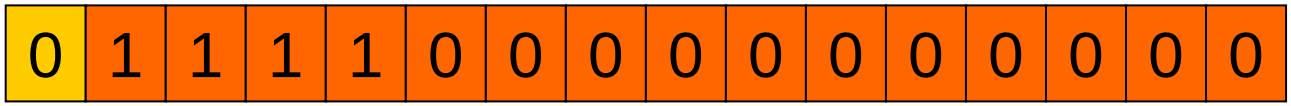
Either we get exactly the right answer for addition, subtraction, or multiplication, or we can detect overflow.

The numbers that we can store are equally spaced.

Disadvantage: **very** limited range of numbers.

Floating point arithmetic

If we wanted to store 15×2^{11} , we would need 16 bits:



Instead, let's agree to code numbers as **two** fixed point binary numbers:

$$z \times 2^{p}, \qquad \text{with } z = 15 \text{ saved as } 01111$$
$$\text{and } p = 11 \text{ saved as } 01011.$$

Now we can have fractions, too:

$$\text{binary } 0.101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} .$$

Floating point arithmetic

Jargon: z is called the **mantissa** or **significand**.
p is called the **exponent.**

$$\pm z \times 2^{p}$$

To make the representation unique (since, for example,

$2 \times 2^{1} = 4 \times 2^{0}$), we **normalize** to make $1 \le z < 2$.

We store d digits for the mantissa, and limit the range of the exponent to $m \le p \le M$, for some integers m and M.

Floating point representation

Example: Suppose we have a machine with d = 5, m = -15, M = 15.

$15 \times 2^{10} = 1111_2 \times 2^{10} = 1.111_2 \times 2^{13}$

mantissa $z = +1.1110$
exponent $p = +1101$

$15 \times 2^{-10} = 1111_2 \times 2^{-10} = 1.111_2 \times 2^{-7}$

mantissa $z = +1.1110$
exponent $p = -0111$

Floating point standard

Up until the mid-1980s, each computer manufacturer had a different choice for d, m, and M, and even a different way to select answers to arithmetic problems.

A program written for one machine often would not compute the same answers on other machines.

The situation improved somewhat with the introduction in 1987 of **IEEE standard** floating point arithmetic.

Floating point standard

On most machines today,

single precision: d = 24, m = -126, M = 127

double precision: d = 53, m = -1022, M = 1023.

(And the representation is 2's complement, not sign-magnitude, so that the number -|x| is stored as 2^d - |x|.)

Floating point addition

Machine arithmetic is more complicated for floating point.

Example: In fixed point, we added 3 + 10.
Here it is in floating point:

$3 = 11$ (binary) $= 1.100 \times 2^1$ $z = 1.100$, $p = 1$
$10 = 1010$ (binary) $= 1.010 \times 2^3$ $z = 1.010$, $p = 11$.

1. Shift the smaller number so that the exponents are equal
   $z = 0.0110$ $p = 11$
2. Add the mantissas
   $z = 0.0110 + 1.010 = 1.1010$, $p = 11$
3. Shift if necessary to normalize.

Roundoff in Floating point addition

Sometimes we cannot store the exact answer.

Example: $1.1001 \times 2^{0} + 1.0001 \times 2^{-1}$

1. Shift the smaller number so that the exponents are equal
   $z = 0.10001 \quad p = 0$
2. Add the mantissas
   0.10001
   $+ 1.1001$
   $= 10.00011, \quad p = 0$
3. Shift if necessary to normalize: 1.000011×2^{1}

But we can only store 1.0000×2^{1}! The error is called **roundoff.**

Underflow, overflow....

Convince yourself that roundoff cannot occur in fixed point.

Other floating point troubles:

Overflow: exponent grows too large.

Underflow: exponent grows too small.

Range of floating point

Example: Suppose that d = 5 and
exponents range between -15 and 15.

Smallest positive normalized number: 1.0000 (binary) x 2^{-15}
(since mantissa needs to be normalized)

Largest positive number: 1.1111 (binary) x 2^{15}

Rounding

IEEE standard arithmetic uses rounding.

Rounding: Store x as r, where r is the machine number closest to x.

An important number: machine epsilon

Machine epsilon is defined to be gap between 1 and the next larger number that can be represented exactly on the machine.

Example: Suppose that d = 5 and exponents range between -15 and 15.

What is machine epsilon in this case?

Note: Machine epsilon depends on d, but does not depend on m or M!

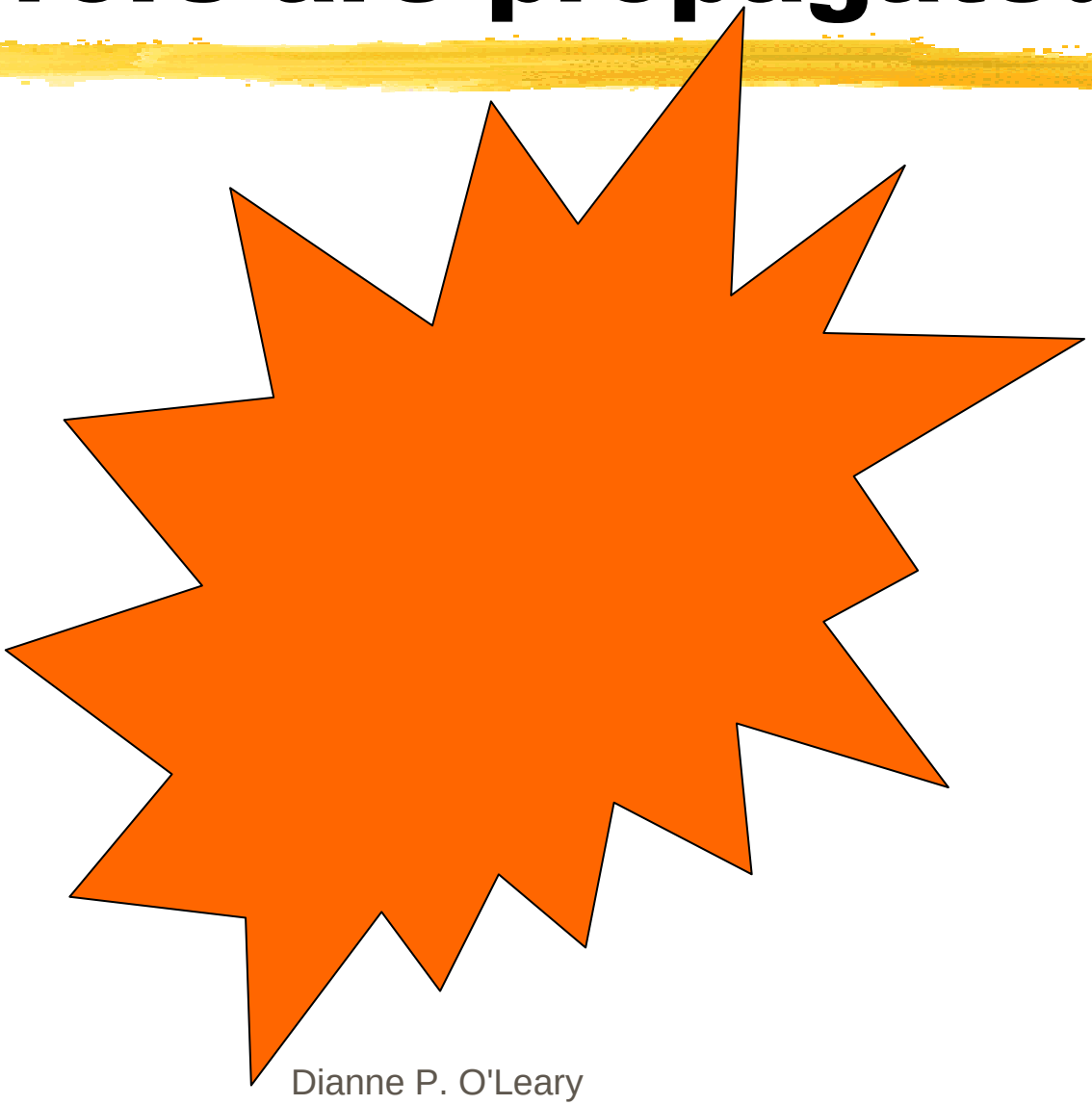
Features of floating point arithmetic

• The numbers that we can store are **not** equally spaced. (Try to draw them on a number line.)

• A wide range of variably-spaced numbers can be represented exactly.

• For addition, subtraction, and multiplication, either we get exactly the right answer or a rounded version of it, or we can detect underflow or overflow.

How errors are propagated



Numerical Analysis vs. Analysis

Mathematical analysis works with computations involving **real** or **complex** numbers.

Computers do not work with these; for instance, they do not have a representation for the numbers π or e or even 0.1 .

Dealing with the finite approximations called **floating point numbers** means that we need to understand **error and its propagation.**

Absolute vs. relative errors

Absolute error in c as an approximation to x:

$$|x - c|$$

Relative error in c as an approximation to nonzero x:

$$\frac{|x - c|}{|x|}$$

Error Analysis

Errors can be magnified during computation.

Example: 2.003×10^0 (suppose $\pm .001$ or .05% error)
$- 2.000 \times 10^0$ (suppose $\pm .001$ or .05% error)

Result of subtraction:

$$0.003 \times 10^0$$

but true answer could be as small as $2.002 - 2.001 = 0.001$,
or as large as $2.004 - 1.999 = 0.005$!

Error Analysis

Errors can be magnified during computation.

Example: 2.003×10^0 (suppose $\pm .001$ or .05% error)
$- 2.000 \times 10^0$ (suppose $\pm .001$ or .05% error)

Result of subtraction:

0.003×10^0 ($\pm .002$ or 200% error if true answer is 0.001)

Catastrophic cancellation, or “loss of significance”

Error Analysis

We could generalize this example to prove a theorem:

When **adding or subtracting**, the bounds on **absolute errors** add.

Error Analysis

What if we multiply or divide?

Suppose x and y are the true values, and X and Y are our approximations to them. If

$$X = x(1 - r) \quad \text{and} \quad Y = y(1 - s)$$

then r is the relative error in x and s is the relative error in y. You could show that

$$\left| \frac{xy - XY}{xy} \right| \leq |r| + |s| + |rs|$$

Error Analysis

Therefore,

- When **adding or subtracting**, the bounds on **absolute errors** add.

- When **multiplying or dividing**, the bounds on **relative errors** add (approximately).

But we may also have additional error -- for example, from chopping or rounding the answer.

Error bounds are useful, but they can be pessimistic. Backward error analysis, discussed later, is an alternative.

Avoiding error build-up

Sometimes error can be avoided by clever tricks.

As an example, consider catastrophic cancellation that can arise when solving for the roots of a quadratic polynomial.

Cancellation example

Example: Find the roots of $x^2 - 56\,x + 1 = 0$.

Usual algorithm: $x_1 = 28 + \text{sqrt}(783) = 28 + 27.982 \quad (\pm .0005)$
$= 55.982 \qquad (\pm .0005)$

$x_2 = 28 - \text{sqrt}(783) = 28 - 27.982 \quad (\pm .0005)$
$= 0.018 \qquad (\pm .0005)$

The **absolute** error bounds are the same, but the **relative** error bounds are 10^{-5} vs. .02!

Avoiding cancellation

Three tricks:

1) **Use an alternate formula.**
   The product of the roots equals the low order term in the polynomial. So

$$x_2 = 1 / x_1 = .0178629 \quad (\pm 2 \times 10^{-7})$$

by our error propagation formula.

Avoiding cancellation

2) **Rewrite the formula.**

$$\text{sqrt}(x + e) - \text{sqrt}(x) = (\text{sqrt}(x+e) - \text{sqrt}(x)) \frac{(\text{sqrt}(x+e) + \text{sqrt}(x))}{(\text{sqrt}(x+e) + \text{sqrt}(x))}$$

$$= \frac{x + e - x}{\text{sqrt}(x+e) + \text{sqrt}(x)} = \frac{e}{\text{sqrt}(x+e) + \text{sqrt}(x)}$$

so $x_2 = 28 - \text{sqrt}(783) = \text{sqrt}(784) - \text{sqrt}(783)$.

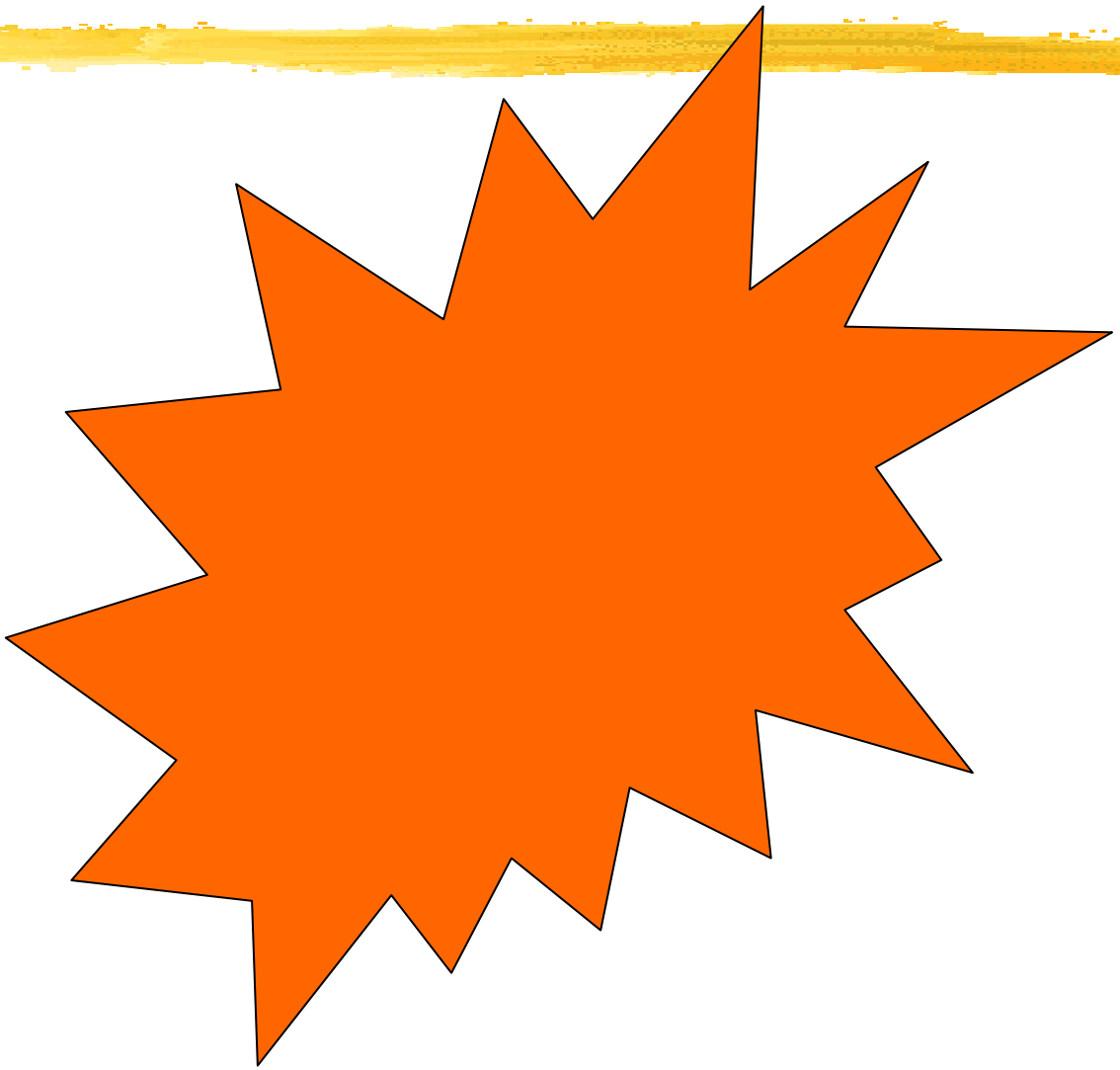
Avoiding cancellation

3) **Use Taylor series.**

Let $f(x) = \text{sqrt}(x)$. Then

$$f(x+a) - f(x) = f'(x)\, a + 1/2\ f''(x)\, a^2 + \ldots$$

How errors are measured



Error analysis

Error analysis determines the cumulative effects of error.

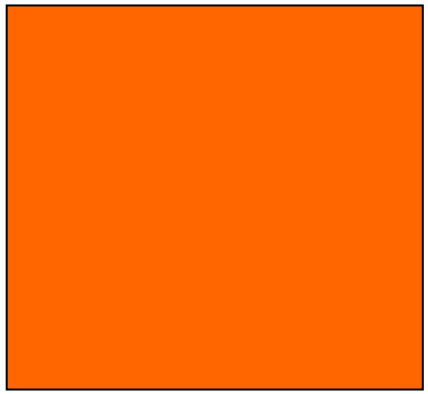
Two approaches:

- Forward error analysis
- Backward error analysis

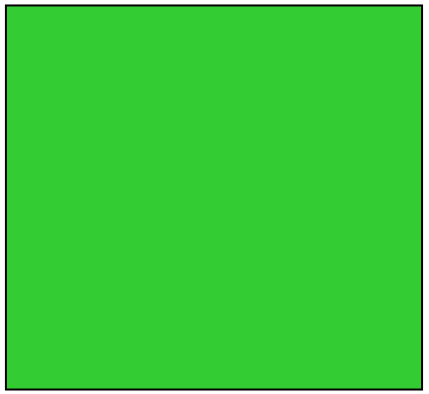
1) Forward error analysis

This is the way we have been discussing.

Find an estimate for the answer, and bounds on the error.



Space of problems

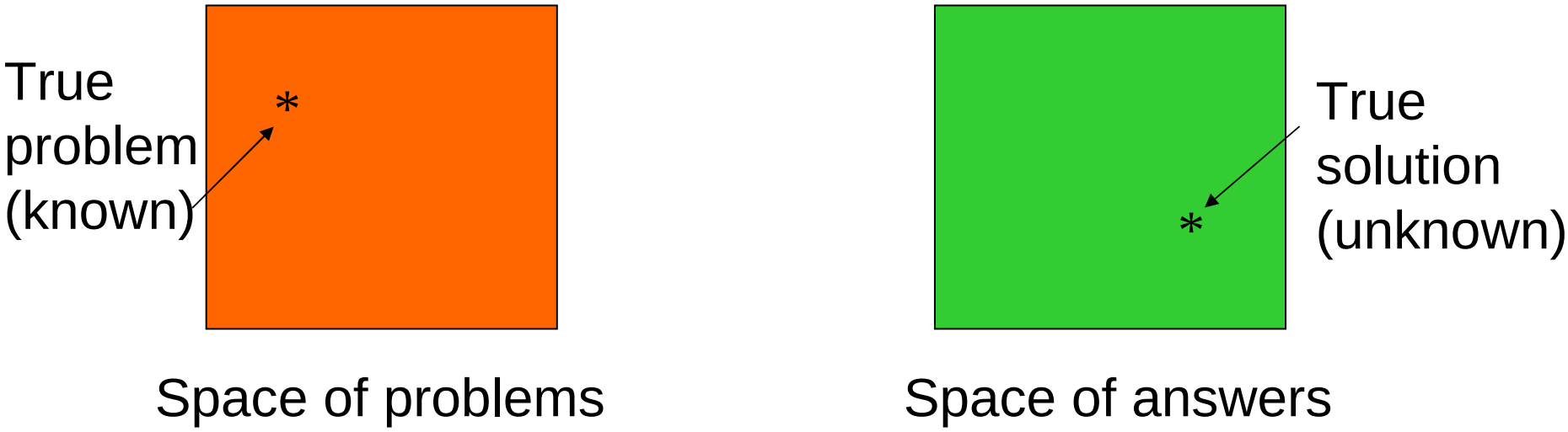


Space of answers

1) Forward error analysis

This is the way we have been discussing.

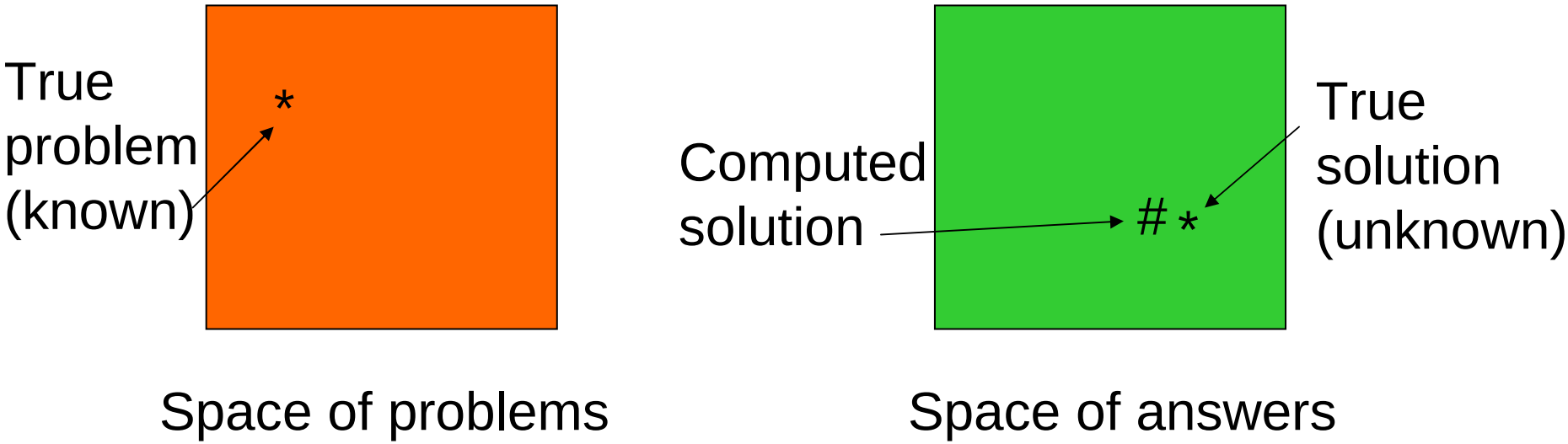
Find an estimate for the answer, and bounds on the error.



1) Forward error analysis

This is the way we have been discussing.

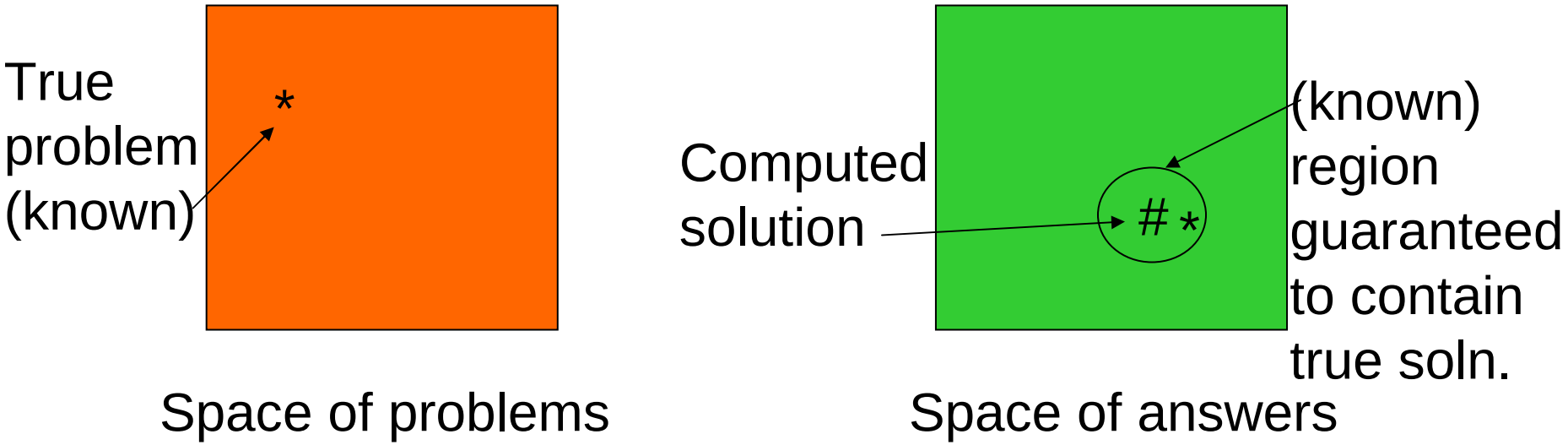
Find an estimate for the answer, and bounds on the error.



1) Forward error analysis

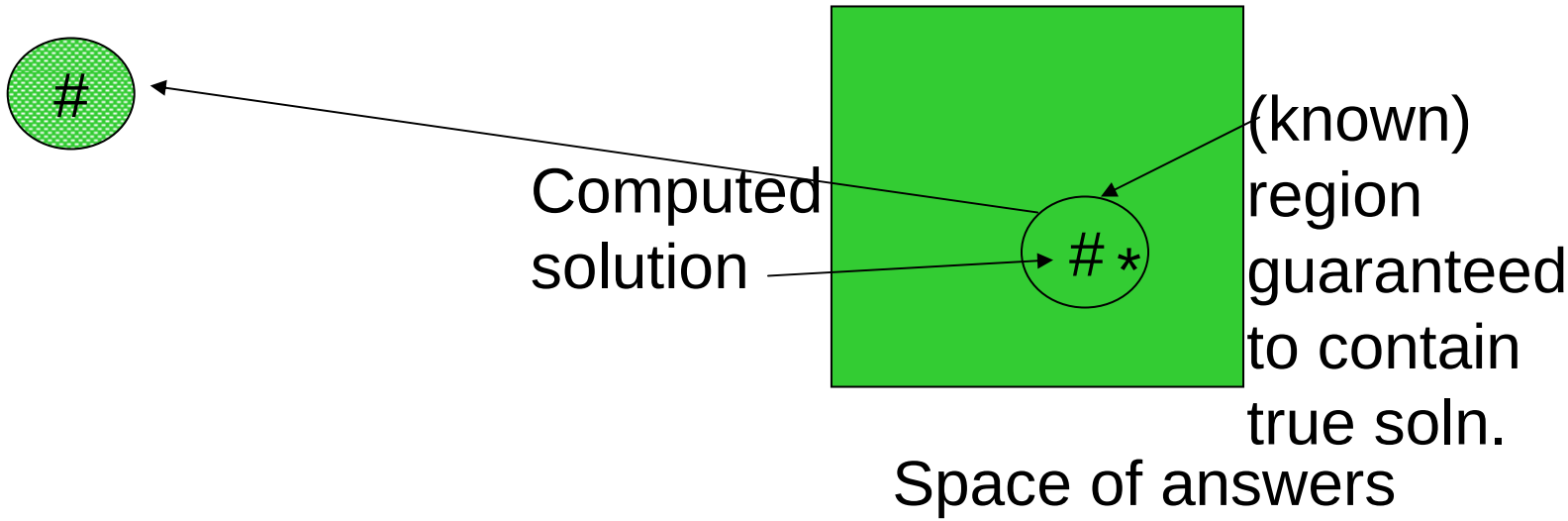
This is the way we have been discussing.

Find an estimate for the answer, and bounds on the error.



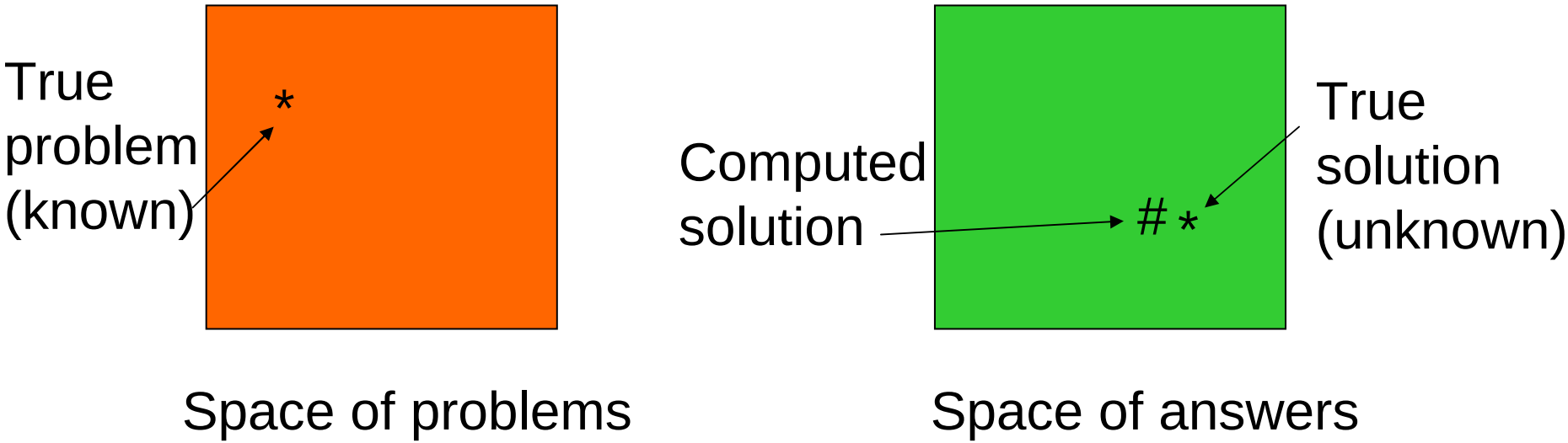
1) Forward error analysis

Report computed solution and location of region to the user.



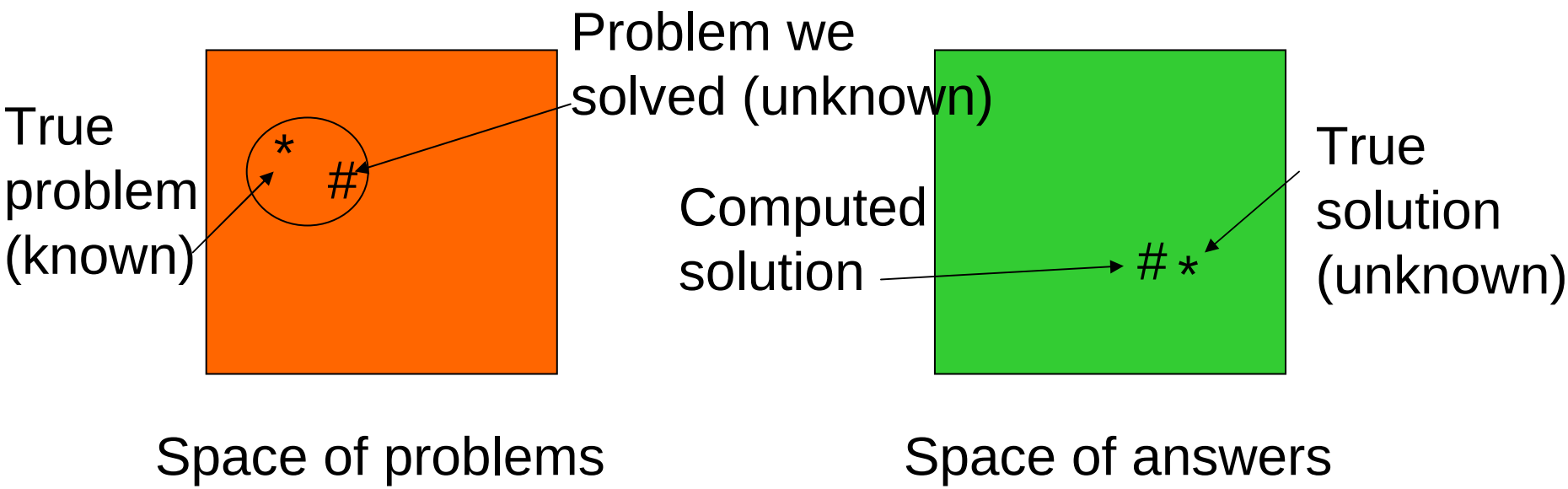
2) Backward error analysis

Given an answer, determine how close the problem actually solved is to the given problem.



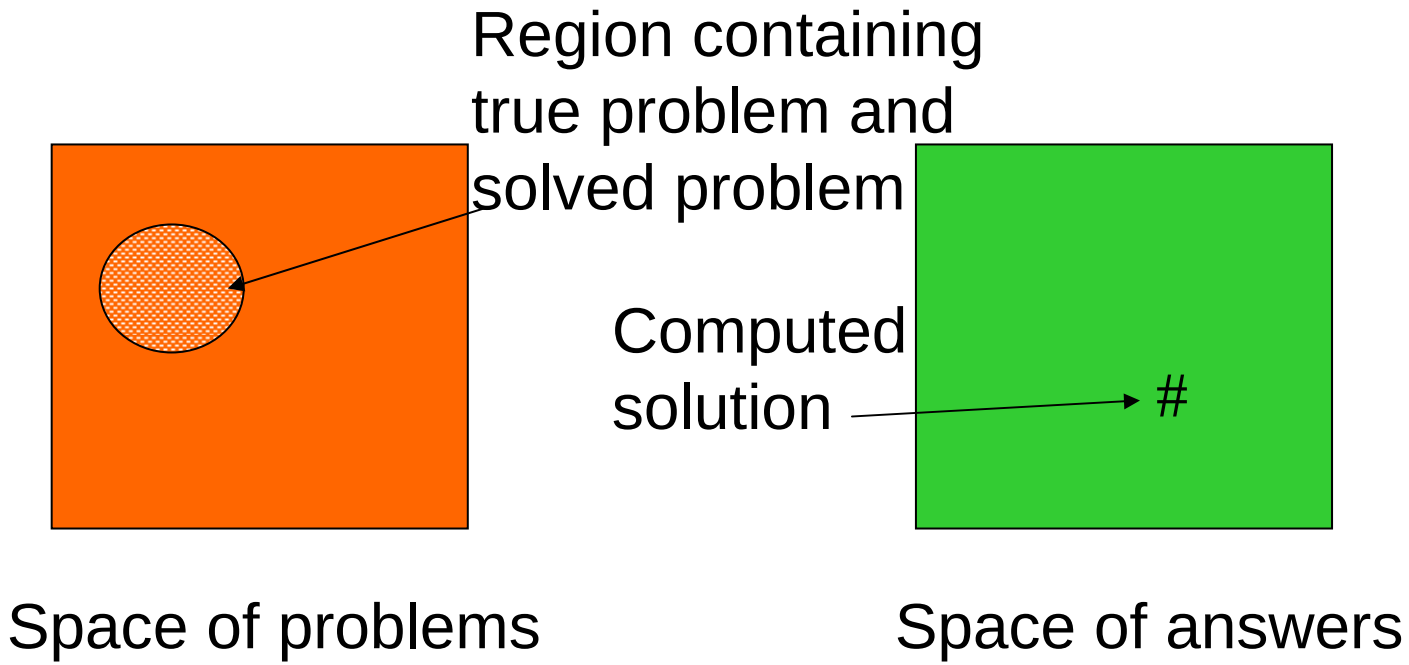
2) Backward error analysis

Given an answer, determine how close the problem actually solved is to the given problem.



2) Backward error analysis

Report computed solution and location of region to the user.



Arithmetic and Error

Summary:

-- how does error arise

-- how machines do arithmetic

  -- fixed point arithmetic
  -- floating point arithmetic

-- how errors are propagated in calculations.

-- how to measure error