MAT 305: Mathematical Computing

Indefinite loops

John Perry

University of Southern Mississippi

Spring 2019

Outline

Outline

Types of loops

- definite
  - # repetitions known at outset
- indefinite
  - # repetitions not known / unknowable at outset

Types of loops

- definite
  - # repetitions known at outset
- indefinite
  - # repetitions not known / unknowable at outset

A lot of languages distinguish these ideas (e.g., Ada, Python)

Types of loops

- definite
  - # repetitions known at outset
- indefinite
  - # repetitions not known / unknowable at outset

A lot of languages distinguish these ideas (e.g., Ada, Python)

C, older C++, & older Java do not really distinguish these; C++11, newer Java introduce some distinction

The `while` command

`while` *condition* `:`
  in-loop statement1
  in-loop statement2
  ...
out-of-loop statement1
...

where

- statements are executed while *condition* remains true
  - statements will *not* be executed if *condition* is false from the get-go
- like definite loops, variables in *condition* can be modified
- unlike definite loops, variables in *condition* **should** be modified

Pseudocode for indefinite loop

while *condition*
    statement1
    statement2
    …
out-of-loop statement 1

Pseudocode for indefinite loop

while *condition*
  statement1
  statement2
  …
out-of-loop statement 1

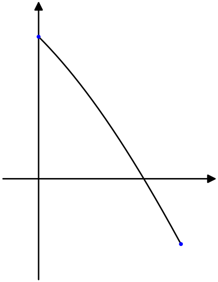
Notice:

- indentation ends at end of loop
- no colon

Example

```
sage:  f = x^10
sage:  while f != 0:
         f = diff(f)
         print f
10*x^9
90*x^8
720*x^7
5040*x^6
30240*x^5
151200*x^4
604800*x^3
1814400*x^2
3628800*x
3628800
0
```

Bisection again!



(need Acrobat Reader to see animation)

This time let's prolong the loop until d digits agree

Pseudocode

algorithm *method_of_bisection*

Pseudocode

algorithm *method_of_bisection*

inputs

f, a continuous function

$a, b \in \mathbb{R}$ such that $a \neq b$ **and** $f(a)$ and $f(b)$ have different signs

d, a positive integer

Pseudocode

algorithm *method_of_bisection*

inputs

f, a continuous function

$a, b \in \mathbb{R}$ such that $a \neq b$ **and** $f(a)$ and $f(b)$ have different signs

d, a positive integer

outputs

$c \in [a, b]$ such that $f(c) \approx 0$ **and** c accurate to d digits

Pseudocode

algorithm *method_of_bisection*

inputs

f, a continuous function

$a, b \in \mathbb{R}$ such that $a \neq b$ **and** $f(a)$ and $f(b)$ have different signs

d, a positive integer

outputs

$c \in [a, b]$ such that $f(c) \approx 0$ **and** c accurate to d digits

do

while the digits of a and b differ through d digits

Let $c = \frac{a+b}{2}$

if $f(a)$ **and** $f(c)$ have the same sign

Let $a = c$ — Interval now $\left(\frac{a+b}{2}, b\right)$

else if $f(a)$ and $f(c)$ have opposite signs

Let $b = c$ — Interval now $\left(a, \frac{a+b}{2}\right)$

else — we must have $f(c) = 0$

return c

return a, rounded to hundredths place

How to check digits?

Round, of course!

How to check digits?

Round, of course! …Oh, really? How far should we round?

Example (Do π and 3.141 agree on first two three digits?)

```
sage: round(pi, 3) == round(3.141, 3)
false
```

How to check digits?

Round, of course! …Oh, really? How far should we round?

Example (Do π and 3.141 agree on first two three digits?)

```
sage: round(pi, 3) == round(3.141, 3)
false
```

Think about it: $\pi \approx 3.1415$ rounds to 3.14*2*

Truncate, instead

3.14159

Truncate, instead

$$3.14159 \longrightarrow 3141.59$$

- Multiply by 10^d

Truncate, instead

$$3.14159 \longrightarrow 3141.59 \longrightarrow 3141$$

- Multiply by 10^d
- Compute the floor (greatest integer function)

Truncate, instead

$$3.14159 \longrightarrow 3141.59 \longrightarrow 3141 \longrightarrow 3.141$$

- Multiply by 10^d
- Compute the floor (greatest integer function)
- Divide by power of 10

Sage code to do this

```
sage:  def trunc(a, d=2):
        a *= 10^d
        a = floor(a)
        return a/10^d
```

Sage code to do this

```
sage:  def trunc(a, d=2):
         a *= 10^d
         a = floor(a)
         return a/10^d
sage:  trunc(pi, 3)
3141/1000
```

Sage code to do this

```
sage: def trunc(a, d=2):
        a *= 10^d
        a = floor(a)
        return a/10^d
sage: trunc(pi, 3)
3141/1000
```

ARGH — How can we fix this?

Sage code to do this

```
sage: def trunc(a, d=2):
        a *= 10^d
        a = floor(a)*1.0
        return a/10^d
sage: trunc(pi, 3)
3141/1000
```

ARGH — How can we fix this?

Introduce a decimal!

Sage code to do this

```
sage:  def trunc(a, d=2):
        a *= 10^d
        a = floor(a)*1.0
        return a/10^d
sage:  trunc(pi, 3)
3141/1000
```

ARGH — How can we fix this?

Introduce a decimal!

```
sage:  trunc(pi, 3)
3.14100000000000
```

Sage code to do this

```
sage:  def trunc(a, d=2):
         a *= 10^d
         a = floor(a)*1.0
         return a/10^d
sage:  trunc(pi, 3)
3141/1000
```

ARGH — How can we fix this?

Introduce a decimal!

```
sage:  trunc(pi, 3)
3.14100000000000
```

Maybe add this to your `calc_utils.sage` script?

(kind of amazed this isn't built in)

Try it!

```
sage:  def method_of_bisection(f, a, b, d=2, x=x):
          f(x) = f
          while trunc(a, d) != trunc(b, d):
```

Try it!

```
sage: def method_of_bisection(f, a, b, d=2, x=x):
        f(x) = f
        while trunc(a, d) != trunc(b, d):
          c = (a + b)/2
```

Try it!

```
sage:  def method_of_bisection(f, a, b, d=2, x=x):
         f(x) = f
         while trunc(a, d) != trunc(b, d):
           c = (a + b)/2
           if f(a)*f(c) > 0:
             a = c
           elif f(a)*f(x) < 0:
             b = c
           else:
             return c
         return round(a,d)
```

Try it!

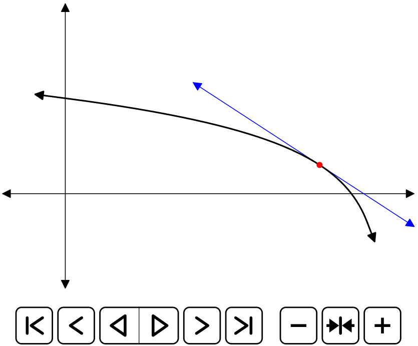
```
sage: def method_of_bisection(f, a, b, d=2, x=x):
        f(x) = f
        while trunc(a, d) != trunc(b, d):
          c = (a + b)/2
          if f(a)*f(c) > 0:
            a = c
          elif f(a)*f(x) < 0:
            b = c
          else:
            return c
        return round(a,d)

sage: method_of_bisection(cos(x)-x,x,0,1)
0.74
```

Outline

Another way of finding a root

- tangent line approximates f
- start close to root? line's root should approximate f's root
- repeat as long as first d digits change



Problem analysis

We need to:

- compute tangent line
- find line's root
- decide if first d digits changed

Problem analysis

We need to:

- compute tangent line
- find line's root
- decide if first d digits changed

How do we decide if first d digits changed?

- `trunc()` again!
- compare current, previous approx's
- *need to remember previous!*

Pseudocode

algorithm *newtons_method*
inputs
  f, a differentiable function
  a, approximation to a root of f
  d, positive number
outputs
  b, "better" approximation to a root of f
do
  let $b = a$
  let $a = b - 1$
  while a, b differ in first d digits
    let $a = b$
    let b be root of line tangent to f at $x = a$
  return b

What are this line
and this one up to?

why?

Sage code

Avoid re-inventing the wheel: re-attach `calc_utils.sage`

Sage code

Avoid re-inventing the wheel: re-attach `calc_utils.sage`

```
sage: def newtons_method(f, a, b, d=2, x=x):
        f(x) = f
        df(x) = diff(f, x)
        b, a = a, a - 1
        while trunc(a, d) != trunc(b, d):
          a = b
          sols = solve(tangent_line(f, a, x), x)
          b = sols[0].rhs()
        return b
```

Sage code

Avoid re-inventing the wheel: re-attach `calc_utils.sage`

```
sage: def newtons_method(f, a, b, d=2, x=x):
        f(x) = f
        df(x) = diff(f, x)
        b, a = a, a - 1
        while trunc(a, d) != trunc(b, d):
          a = b
          sols = solve(tangent_line(f, a, x), x)
          b = sols[0].rhs()
        return b
```

works great, except:

```
sage: newtons_method(cos(x) - x, 0.5, 4)
1/75485362136393*(75485362136393*cos(57008237648741/7548
```

Sage code

Avoid re-inventing the wheel: re-attach `calc_utils.sage`

```
sage: def newtons_method(f, a, b, d=2, x=x):
        f(x) = f
        df(x) = diff(f, x)
        b, a = a, a - 1
        while trunc(a, d) != trunc(b, d):
          a = b
          sols = solve(tangent_line(f, a, x), x)
          b = sols[0].rhs()
        return b
```

works great, except:

```
sage: newtons_method(cos(x) - x, 0.5, 4)
1/75485362136393*(75485362136393*cos(57008237648741/754
```

Sage solves for the line's root exactly!

Sage code

How can we get around it?

Sage code

How can we get around it?

```
sage:  def newtons_method(f, a, b, d=2, x=x):
         f(x) = f
         df(x) = diff(f, x)
         b, a = a, a - 1
         while trunc(a, d) != trunc(b, d):
           a = b
           sols = solve(tangent_line(f, a, x), x)
           b = float(sols[0].rhs())
         return b
sage:  newtons_method(cos(x) - x, 0.5, 4)
0.7390851332151602
```

Outline

Gaussian integers

Definition

A **Gaussian integer** has the form $a+bi$ where $a,b\in\mathbb{Z}$ and $i^2=-1$.

Examples

$$7, \quad 2+3i, \quad -3+2i$$

but *definitely not*

$$\frac{3}{2}, \quad \pi, \quad \frac{1}{3}-i\frac{5}{2}.$$

Gaussian integers form a ring

$$(a+bi)\pm(c+di)=(a\pm c)+i(b\pm d)$$

$$(a+bi)(c+di)=(ac-bd)+i(ad+bc)$$

Gaussian integers form a ring

$$(a+bi)\pm(c+di)=(a\pm c)+i(b\pm d)$$

$$(a+bi)(c+di)=(ac-bd)+i(ad+bc)$$

Can we also *divide* by Gaussian integers?

Analyze the problem

What does division mean, anyway?

Analyze the problem

What does division mean, anyway?

Repeated subtraction.

Example

$$40 = 13 \times 3 + 1$$

Extend the meaning

Integer division:

Subtract until the remainder's size is less than the divisors's.

Gaussian integer division:

Subtract until the remainder's size is less than the divisor's.

Extend the meaning

Integer division:

Subtract until the remainder's size is less than the divisors's.

Gaussian integer division:

Subtract until the remainder's size is less than the divisor's.

What do we mean by "remainder's size"?

Size of a number

In $\mathbb{Z}$, "size" is called **absolute value:**

$$|-5| = |5| = 5$$

Size of a number

In $\mathbb{Z}$, "size" is called **absolute value:**

$$|-5| = |5| = 5$$

In $\mathbb{Z}[i]$, "size" is called **(Euclidean) norm:**

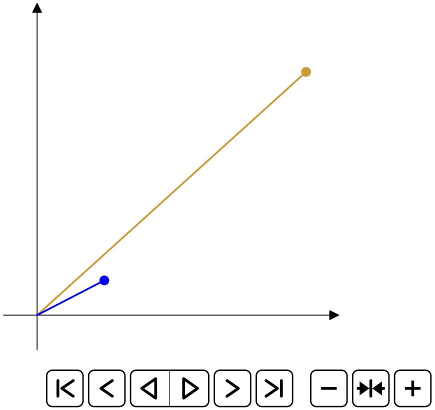
$$||a + bi|| = a^2 + b^2$$

Example

$||2 + 3i|| = 2^2 + 3^2 = 13$

Geometrically

Divide $8+7i$ by $2+i$:



A wrench in the system…

We found that

$$8+7i=5\times(2+i)+(-2+2i)$$

but…

A wrench in the system…

We found that

$$8+7i=5\times(2+i)+(-2+2i)$$

but…

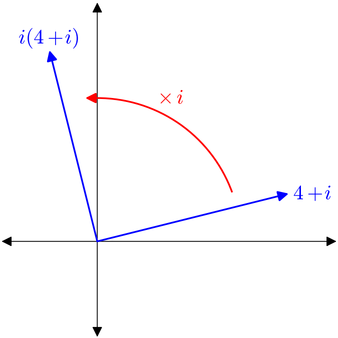
$$\|-2+2i\|=4+4 \quad > \quad 4+1=\|2+i\|$$

The distance is larger than we'd like!

Can we do better?

Recall Programming #4 in "Pretty Pictures":

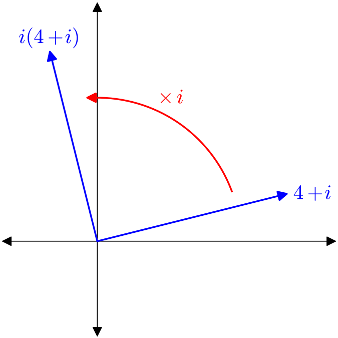
Multiplying a complex number by i rotates it 90° counterclockwise.



Can we do better?

Recall Programming #4 in “Pretty Pictures”:

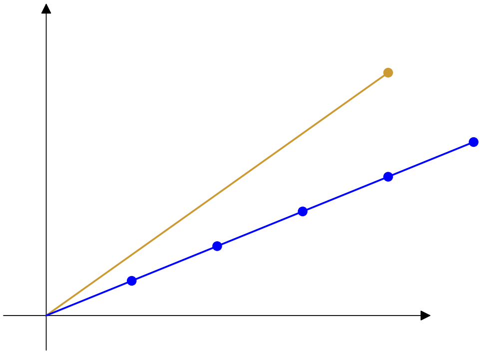
Multiplying a complex number by i rotates it 90° counterclockwise.



...so we *should* get closer if we add *imaginary* multiples of $2+i$.

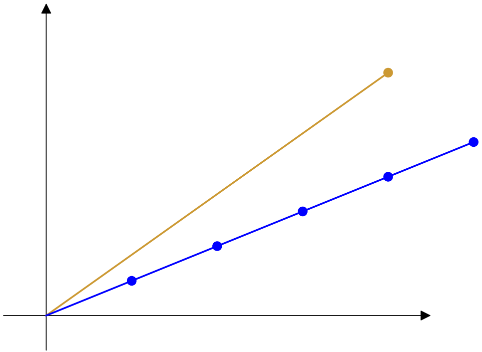
Geometrically (again)

Divide $8+7i$ by $2+i$ *completely*:



Geometrically (again)

Divide $8+7i$ by $2+i$ *completely*:



We have $8+7i = (5+i)(2+i)+(-1)$

Pseudocode

algorithm *gaussian_reduction*
inputs
  $z, d \in \mathbb{Z}[i]$ such that $||d|| > 0$
outputs
  $q, r \in \mathbb{Z}[i]$ such that $z = qd + r$ and $||r|| < ||d||$
do
  let $r = z$, $q = 0$
  while $z - qr$ grows smaller
    add/subtract 1 to/from q, as appropriate
  while $z - qr$ grows smaller
    add/subtract i to/from q, as appropriate
  return q, r

Sage code

```
def divide_gaussian_integers(z, d):
    r, q = z, 0
```

Sage code

```
def divide_gaussian_integers(z, d):
    r, q = z, 0
    # which real way to step?
    if norm(r - d) < norm(r):
        s = 1
    else:
        s = -1
```

Sage code

```
def divide_gaussian_integers(z, d):
    r, q = z, 0
    # which real way to step?
    if norm(r - d) < norm(r):
        s = 1
    else:
        s = -1
    # loop to step
    while norm(r - s*d) < norm(r):
        q = q + s
        r = r - s*d
```

Sage code

```
def divide_gaussian_integers(z, d):
    r, q = z, 0
    # which real way to step?
    if norm(r - d) < norm(r):
        s = 1
    else:
        s = -1
    # loop to step
    while norm(r - s*d) < norm(r):
        q = q + s
        r = r - s*d
    # which imaginary way to step?
    if norm(r - I*d) < norm(r):
        s = I
    else:
        s = -I
```

Sage code

```
def divide_gaussian_integers(z, d):
    r, q = z, 0
    # which real way to step?
    if norm(r - d) < norm(r):
        s = 1
    else:
        s = -1
    # loop to step
    while norm(r - s*d) < norm(r):
        q = q + s
        r = r - s*d
    # which imaginary way to step?
    if norm(r - I*d) < norm(r):
        s = I
    else:
        s = -I
    # loop to step
    while norm(r - s*d) < norm(r):
        q = q + s
        r = r - s*d
    return q, r
```

Example

```
sage: divide_gaussian_integers(8 + 7*I, 2 + I)
(I + 5, -1)
```

Outline

Summary

Two types of loops

- definite: n repetitions known at outset
  - **for** $c \in C$
    - collection C of n elements controls loop
    - don't modify C
- indefinite: number of repetitions not known at outset
  - **while** *condition*
    - Boolean *condition* controls loop

Awesome mathematics!

- Newton's method
- Gaussian integers
  - division, too!