Nesta aula…

Conteúdo

1 Método de bissecções

Resolução numérica de equações

Problema: encontrar uma solução x^* da equação

$$x^3 - 5x + 2 = 0$$

Equivalente: encontrar zeros da função

$$f(x) = x^3 - 5x + 2$$

- Polinómio do 3º grau: não tem uma fórmula resolvente simples
- Podemos obter *aproximações* $x_1, x_2, \ldots$ tais que $\lim_{n\to\infty} x_n = x^*$
- Paramos quando o *erro* $|x^* - x_n|$ for aceitável

Método de bissecções

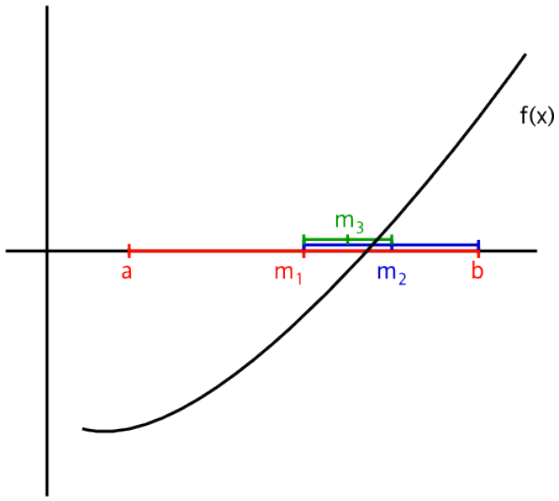
Pré-condições:

- f contínua em $[a, b]$
- $f(a) \times f(b) < 0$ *(*$f(a)$ *e* $f(b)$ *têm sinais opostos)*
- f tem uma raiz única em $[a, b]$

Método recursivo:

1. calculamos o ponto médio $m = (a + b)/2$
2. se $f(a) \times f(m) < 0$: a raiz está em $[a, m]$
3. caso contrário: a raiz está em $[m, b]$
4. repetir com o novo intervalo encontrado

Método de bissecções



Método de bissecções recursivo

```
def bissect(f, a, b, n):
  "Efectuar n bissecções de f(x)=0 em [a,b]."
  # calcula o ponto médio
  m = (a+b)/2.0
  # terminou?
  if n==0:
     return m   # ponto médio
  # senão, parte o intervalo ao meio
  if f(a)*f(m) < 0:
     return bissect(f, a, m, n-1)
  else:
     return bissect(f, m, b, n-1)
```

Execução

```
# equação x**3-5*x+2==0
def eq(x):
   return x**3-5*x+2

# zero em [0,1] com 5 iterações
>>> bissect(eq, 0.0, 1.0, 5)
0.421875
# zero em [0,1] com 10 iterações
>>> bissect(eq, 0.0, 1.0, 10)
0.41455078125
```

```
# verificação
>>> eq(0.41455078125)
-0.0015123802004382014
```

Eliminar a recursão

- chamadas recursivas apenas no *final* da função
- podem ser substituídas por *iteração* com um ciclo `while`
- vantagem: reduz o uso de pilha de execução (*stack*)

Método de bissecções iterativo

```
def bissect(f, a, b, n):
  "Efectua n bissecções de f(x)=0 em [a,b]."
  while n>0:
    m = (a+b)/2.0  # calcula o ponto médio
    # parte o intervalo ao meio
    if f(a)*f(m) < 0:
       b = m
    else:
       a = m
    n = n-1
  # fim do ciclo
  # resultado é o ponto médio final
  return m
# fim do programa
```

Exercício

Seja $x^\star$ a raíz exacta e x a aproximação retornada pela função. O erro da aproximação é $|x^\star - x|$.

Exercício Escreva uma função

`bissect2(f,a,b,eps)`

que retorna uma aproximação da raíz da equação $f(x) = 0$ com um erro não superior a `eps`.

Optimizar a implementação

- reduzir o custo computacional do algoritmo
- em cada iteração: calcula f três vezes
- podemos fazer melhor:

1. guardar $f(a)$, $f(b)$ e $f(m)$ em fa, fb e fm
2. actualizamos as variáveis de forma a manter *invariantes* as condições:

$$\begin{cases} fa = f(a) \\ fb = f(b) \\ fm = f(m) \end{cases}$$

Método de bissecções optimizado

```
def bissect(f, a, b, n):
  "Efectua n bissecções de f(x)=0 em [a,b]."
  fa = f(a)         # valores de f nos extremos
  fb = f(b)
  while n>0:
    m = (a+b)/2.0   # calcula o ponto médio
    fm = f(m)       # valor de f no ponto médio
    ⋮

    ⋮
    # parte o intervalo ao meio
    if fa*fm < 0:
       b = m
       fb = fm  # manter invariante
    else:
       a = m
       fa = fm  # manter invariante
    n = n-1     # menos uma iteração
  # fim do ciclo
  # resultado é o ponto médio final
  return m
# fim do programa
```

Análise do método optimizado

- em cada iteração: calcula f uma vez
- reduzimos o custo de calcular f em 50%
- vantagem: f pode ser uma expressão matemática complexa
- usamos apenas três variáveis extra

Sumário

- começar pela *formulação* abstracta do problema
- fazer uma implementação *simples* e *correcta*
- *refinar* a implementação passo-a-passo

2 Cadeias de caracteres

Cadeias de caracteres

- compostas por caracteres individuais
- podemos tratá-las como um entidade única
- podemos também aceder aos caracteres individuais

```
>>> fruta = 'Banana'
>>> fruta[0]
'B'
>>> fruta[1]
'a'
>>> fruta[2]
'n'
>>> len(fruta)
6
```

Índices

`str` $\longrightarrow$	`'B`	`a`	`n`	`a`	`n`	`a'`
	0	1	2	3	4	5

- índices: `0`, `1`, `2`, `3`, `4`, `5`
- em geral: `0`, `1`, ..., `len(str)-1`
- caracteres: `str[0]`, `str[1]`, ...
- último caracter: `str[len(str)-1]`
- penúltimo caracter: `str[len(str)-2]`

Percorrer uma cadeia

Usando um ciclo `while`

```
# invariante: 0<=i<=len(fruta)-1
i = 0
while i<len(fruta):
  letra = fruta[i]
  print letra
  i = i + 1
```

Strings: o que é permitido usar nesta disciplina

Em problemas com "strings" o aluno apenas poderá usar operações elementares (a não ser que o enunciado diga o contrário)

- Acesso a um índice de cada vez; por exemplo `fruta[i]` mas não `fruta[: i]`
- Comparações envolvendo os caracteres que estão nos índices (um caracter de cada vez); por exemplo "`fruta[i]` == "`b`" mas não `fruta` == "`banana`"
- A função `len()`; por exemplo `len(fruta)` mas não `lower(fruta)`

Percorrer uma cadeia (2)

Idioma: o ciclo `for`

```
for letra in fruta:
  print letra
```

- evita a utilização explícita do índice
- generalização: percorrer *sequências* (mais tarde)