9 Alg. de Horner: Polynomial evaluation

Consider

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

We wish to compute $p(k)$ for a given $k \in \mathbb{R}$ minimizing the number of operations. Directly computing $a_0 + a_1 k_1 + \ldots$ leads to n sums. The ith term requires computing k^i, which means i product operations, for a totall of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ products. The total number of operations is then

$$\Theta = n + n(n+1)/2$$

The associated complexity is $\mathcal{O}(n^2)$.

Horner's method consists of re-writing $p(x)$ so that the number of products is reduced. One writes

$$p(x) = a_0 + x b_0$$

where $b_{n-1} = a_n$ and for $0 \leq i < n - 1$:

$$b_{i-1} = a_i + x b_i$$

Let $p(x) = 3 + 5x - 4x^2 + 0x^3 + 6x^4$, giving $n = 4$. Then $b_3 = 6$ and

$$b_2 = a_3 + x b_3 = 6x, \qquad b_1 = a_2 + x b_2 = -4 + x(6x),$$
$$b_0 = a_1 + x b_1 = 5 + x(-4 + x(6x))$$

This finally gives

$$p(x) = 3 + x b_0 = 3 + x(5 + x(-4 + x(6x)))$$

Here, one must perform n sums again but only n products. Thus, there are $\Theta = n + n = 2n$ operations, giving a complexity of $\mathcal{O}(n)$ (in the operation space). See the algorithm below:

$$
\begin{array}{l}
\textbf{input } n; a_i, i = 0, \dots, n; x \\
b_{n-1} \leftarrow a_n \\
\textbf{for } i = n-2 \textbf{ to } i = 0 \\
\qquad b_i = a_{i+1} + x * b_{i+1} \\
\textbf{od} \\
y \leftarrow a_0 + x * b_0 \\
\textbf{return } y
\end{array}
$$

It is easy to see in this code that the **for** loop performs $n-1$ iterations, in each of which a single sum and a single product are computed. The nth sum and nth product are performed in the computation of y, the final result.

A more polished version includes the last computatoin (the one in the assignment of y) within the loop and makes no use of indexes:

$$
\begin{array}{l}
\textbf{input } n; a_i, i = 0, \dots, n; x \\
b \leftarrow a_n \\
\textbf{for } i = n-2 \textbf{ to } i = -1 \\
\qquad b = a_{i+1} + x * b \\
\textbf{od} \\
\textbf{return } b
\end{array}
$$

In Python,

```
def horner(coefs, x):
  n = len(coefs)-1
  b = coefs[n]

  for i in reversed(range(-1, n-1)):
    b = coefs[i+1] + x*b

  return b
```

It is trivial to adapt the code so that it returns the coefficients $b_0, \dots, b_{n-1}$ and not the final result, if needed.

10 Error

Let $r, \overline{r}$ be two real numbers s.t. the latter is an approximation of the first. We define the **error** of the approximation to be $r - \hat{r}$, and

$$\Delta r = |r - \overline{r}|\,, \qquad \delta r = \frac{\Delta r}{|r|}$$

With r unknown the strategy is to work with a known bound of r.

11 Non-linear equations

The general problem is to find members of the set $\mathcal{R}_f$ of roots of $f \in \mathbb{R} \to \mathbb{R}$. The numerical strategy is to iteratively approximate some $r \in \mathcal{R}_f$ until some pre-established threshold in the error of approximation is met.

More formally, the numerical strategy produces a sequence $\{x_k\}_{k \in \mathbb{N}}$ which satisfies

- $\lim_{k\to\infty} \{x_k\} = r$ for some $r \in \mathcal{R}_f$
- Either $e(x_k) < e(x_{k-1})$ or, more strongly, $\lim_{k\to\infty} e(x_k) = 0$, where $e(x_k)$ is some appropriate measure of the error of approximation.

11.1 Bisection

A very simple procedure: if a root exists in $[a, b]$, it iteratively shrinks $[a, b]$ in halves (keeping the halves which contain the root) until the interval is of sufficiently small length.

[Intermediate value] If f is continuous in $[a, b]$ and $f(a)f(b) < 0$, then $\exists r \in \mathcal{R}_f$ s.t. $r \in [a, b]$.

Assume f is continuous. A root exists in $[a, b]$ if $f(a)f(b) < 0$ (**Theorem 1**). If that is the case, the midpoint $(a+b)/2$ is taken as the approximation x_0. It is also trivial to observe that x_0 is *at most* at a distance of $(b-a)/2$ from the real root, so $e_0 = |x_0 - r| \leq (b-a)/2$.

If $f(x_0) = 0$ the procedure must end because a root was found. Otherwise, sufficies to find which half of the interval contains a root computing $f(a)f(c)$ and, if needed, $f(c)f(b)$.

The iterations may stop after reaching a maximum number of steps, when $|f(c)|$ is sufficiently close to zero, or when the error bound $|e_k| \leq (b_k - a_k)/2$ (where $[a_k, b_k]$ is the interval of this iteration) is sufficiently small.

(!) The algorithm not always converges. Take $f(x) = 1/x$. Clearly, it has no root. Yet setting $a = -1, b = 1$ in the initial iteration falsely passes the test. (The problem obviously is that f is not continuous in $[-1, 1]$.) If one sets

Input : a, b, δ, M, f
Output : Tupla de la forma: $(r, \text{cota de error})$
$f_a \leftarrow f(a)$
$f_b \leftarrow f(b)$

if $f_a * f_b > 0$
    return ?
fi

for $i = 1$ **to** $i = M$ **do**
    $c \leftarrow a + (b - a)/2$
    $f_c \leftarrow f(c)$
    if $f_c = 0$ **then**
        return $(c, 0)$
    fi
    $\epsilon = \dfrac{b - a}{2}$
    if $\epsilon < \delta$ **then**
        break
    fi
    if $f_a * f_c < 0$ **then**
        $b \leftarrow c$
        $f_b = f(b)$
    else
        $a \leftarrow c$
        $f_a = f(a)$
    fi
od
return (c, ϵ)

```
def bisection(f : callable, a : float, b : float, delta : float, M : int):

  s, e = f(a), f(b) # function values at (s)tart, (e)nd of interval

  if s*e > 0:
    raise ValueError("Interval [a, b] contains no root.")

  for i in range(M):

    c = a + (b-a)/2
    m = f(c) # value of f at (m)idpoint

    if m == 0:
      return c, 0

    e = (b-a)/2
    if e < delta:
      return c, e

    if s*m < 0:
      b = c
      e = f(b)
    else:
      a = c
      s = f(a)

  return c, e
```

If $\{[a_i, b_i]\}_{i=0}^{\infty}$ are the intervals generated by the bisection method on iterations $i = 0, 1, \ldots$, then:

1. $\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$ is a member of $\mathcal{R}_f$.
2. If $c_n = \frac{1}{2}(a_n + b_n), r = \lim_{n\to\infty} c_n$, then $|r - c_n| \leq \frac{1}{2^{n+1}}(b_0 - a_0)$

Proof. (1) It is clear that $a_i \leq a_{i+1}$ and $b_i \geq b_{i+1}$, since the interval on each iteration shrinks in one direction.

$\therefore a_n, b_n$ are monotonous.

But clearly a_n is bounded by b_0 and b_n is bounded by a_0.

$\therefore a_n, b_n$ are monotonous and bounded.

$\therefore$ Their limits exist.

It is also clear that the interval shrinks to half its size on each iteration:

$$b_n - a_n = \frac{1}{2}(b_{n-1} - a_{n-1}), \qquad n \geq 1 \tag{1}$$

By recurrence on (1),

$$b_n - a_n = \frac{1}{2^n}(b_0 - a_0), \qquad n \geq 0 \tag{2}$$

Then

$$\lim_{n\to\infty} a_n - \lim_{n\to\infty} b_n = \lim_{n\to\infty}(a_n - b_n) = \lim_{n\to\infty} \frac{1}{2^n}(b_0 - a_0) = 0 \tag{3}$$

$\therefore \lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$.

Since the limit of a_n, b_n exists and f is by assumption continuous, the composition limit theorem applies and:

$$\begin{aligned}
&\lim_{n\to\infty}(f(a_n)\cdot f(b_n)) \\
&= \lim_{n\to\infty} f(a_n)\cdot \lim_{n\to\infty} f(b_n) && \{\text{Product of limits}\} \\
&= f\left(\lim_{n\to\infty} a_n\right)\cdot f\left(\lim_{n\to\infty} b_n\right) && \{\text{Composition limit theorem}\} \\
&= [f(r)]^2 && \left\{r = \lim_{n\to\infty} a_n\right\}
\end{aligned} \tag{4}$$

The invariant of the algorithm is $f(a_n)f(b_n) < 0$. But due to the last result,

$$\lim_{n\to\infty} f(a_n)f(b_n) \leq 0 \iff [f(r)]^2 \leq 0 \iff f(r) = 0$$

$\therefore r = \lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$ is a root.

(2) Follows directly from result (2)

$$
\begin{aligned}
|r - c_n| &= \left| r - \frac{1}{2}(b_n - a_n) \right| \\
&\leq \left| \frac{1}{2}(b_n - a_n) \right| \\
&= \left| \frac{1}{2^{n+1}}(b_0 - a_0) \right| && \{\text{Result (2)}\}
\end{aligned}
$$

11.2 Newton's method

Taylor: repasito. El desarrollo de una f suficientemente diferenciable alrededor de un punto r espa

$$f(x) = f(r) + f'(r)(x-r) + \frac{f''(r)}{2!}(x-r)^2 + \ldots + \frac{f^{(n)}(r)}{n!}(x-r)^n + R_n(x)$$

donde $R_n(x)$ es el resto.

Usualmente, queremos tomar $r = x + h$, donde x es una aproximación de r y h el error de aproximación. Entonces es provechoso expandir $f(r)$ alreededor de su estimación x:

$$f(r) = f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \ldots + \frac{f^{(n)}(x)}{n!}h^n + R_n(h)$$

Esto es **recontra** útil porque nos dice cuánto se diferencia $f(r)$ de nuestra aproximación $f(x)$ (pues expresa $f(r)$ como $f(x)$ más algo).

Usualmente r, h son desconocidos pero h puede acotarse.

Assume $r \in \mathcal{R}_f$ and $r = x + h$, with x an approximation of r and h its error. Assume f'' exists and is continuous in some I around x s.t. $r \in I$. What we explained on Taylor expansions around a point gives:

$$0 = f(r) = f(x+h) = f(x) + f'(x)h + \mathcal{O}(h^2)$$

If x is sufficiently close to r, h is small and h^2 even smaller, so that $\mathcal{O}(h^2)$ is unconsiderable:

$$0 \approx f(x) + hf'(x)$$

Therefore,

$$h \approx -\frac{f(x)}{f'(x)} \tag{5}$$

From this follows that $r = x + h$ is approximated by

$$r \approx x - \frac{f(x)}{f'(x)}$$

Since the approximation in (5) truncated the terms of $\mathcal{O}(h^2)$ complexity, this new approximation is closer to r than x originally was. In other words, $x - f(x)/f'(x)$ is a better approximation to r than x itself.

Thus, if x_0 is an original approximation, we can define

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{6}$$

to produce a sequence of approximations. This is the fundamental idea of Newton's method.

Input: x_0, M, δ, ϵ;
$v \leftarrow f(x_0)$
if $|v| < \epsilon$ **then return** x_0 **fi**
for $k = 1$ **to** $k = M$ **do**
    $x_1 \leftarrow x_0 - \dfrac{v}{f'(x_0)}$
    $v \leftarrow f(x_1)$
    if $|x_1 - x_0| < \delta \vee v < \epsilon$ **then**
        return x_1
    fi
    $x_0 \leftarrow x_1$
od
return x_0

The predicate $|x_1 - x_0| < \delta$ checks whether our algorithm is adjusting x in a negligible degree. If that is the case, we should stop.

11.2.1 Error in Newton's method