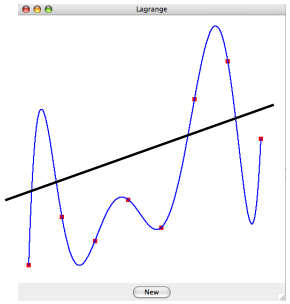
OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

http://deptinfo.unice.fr/~roy

Cours n°7

Polynômes, graphisme, etc



p. 334-335

Représentation des polynômes

• Si les polynômes ont beaucoup de coefficients ≠ 0, on opte pour une représentation **dense** avec des **vecteurs** [les *tableaux* de Java] :

$2x^4 - 7x^3 - 3x + 5$ ➟ `#(5 -3 0 -7 2)`

• Si au contraire les polynômes ont beaucoup de coefficients = 0, on opte pour une représentation **creuse** avec des listes :

$2x^{100} - 3$ ➟ `((2 100) (-3 0))`

un "monôme"
(coefficient exposant)

• Nous allons travailler avec des représentations **creuses** en **puissances décroissantes** !

Une boîte à outils "polynômes"

• Le **polynôme nul** sera représenté par la liste vide.

```
(define poly0 '())
(define poly0? null?)
```

• Un **polynôme non nul** sera représenté par une liste :

$((c_N\ e_N)\ (c_{N-1}\ e_{N-1})\ \ldots)$

en puissances décroissantes, où e_N est le degré.

• Le module ***string2poly.rkt*** fournit deux fonctions pour lire ou écrire un polynôme :

```
> (require "string2poly.rkt")
> (define test (string->poly "7x^3-x^8+5-2x"))
> test
((-1 8) (7 3) (-2 1) (5 0))          ; tri automatique !
> (poly->string test)
"-x^8+7x^3-2x+5"
```

• Un **monôme** n'est pas un polynôme !

```
(define (monome c e)          ; (c e)
  (list c e))

(define (polynome c e)        ; ((c e))
  (list (list c e)))          ; un polynôme est une liste de monômes
```

• Les fonctions `degre` et `coeff` prennent un monôme ou un polynôme :

```
(define (degre mp)            ; p = ((cN eN) (cN-1 eN-1) ...), retourne N
  (cond ((poly0? mp) -inf.0)              ; bof...
        ((pair? (car mp)) (cadar mp))     ; polynôme
        (else (cadr mp))))                ; monôme

(define (coeff mp)            ; retourne cN
  (cond ((poly0? mp) 0)
        ((pair? (car mp)) (caar mp))      ; polynôme
        (else (car mp))))                 ; monôme
```

- Les fonctions `tete` et `reste` sont des abstractions de `car` et `cdr` :

```
(define (tete p)              ; le monôme de tête (cN eN)
  (if (poly0? p)
      (error "Pas de monôme de tête pour le polynôme nul !")
      (car p)))

(define (reste p)             ; le polynôme privé de (cN eN)
  (if (poly0? p)
      (error "Pas de reste pour le polynôme nul !")
      (cdr p)))
```

- Pour efficacité, les opérations sur les polynômes sont dans le type abstrait et utilisent les listes. Définissons p et q pour les tests :

```
> (define p (string->poly "2x^5+x^4+2x^3-2x+3"))
> (define q (string->poly "x^4-2x^3+7x^2-x"))
> p
((2 5) (1 4) (2 3) (-2 1) (3 0))
> q
((1 4) (-2 3) (7 2) (-1 1))
```

- Commençons par la **loi externe** `(poly*ext k p)` :

```
(define (poly*ext k p)          ; num × polynôme → polynôme
  (if (= k 0)
    poly0
    (map (λ (m) (monome (* k (coeff m)) (degre m))) p)))
```

```
> (poly->string (poly*ext 2 p))
"4x^5+2x^4+4x^3-4x+6"
```

- **L'addition** `(poly+ p q)` tient compte des puissances décroissantes, et de l'absence de coefficients nuls !

```
> (poly->string (poly+ p q))
"2x^5+2x^4+7x^2-3x+3"
```

cf TP !

- La **soustraction** `(poly- p q)` se déduit de `poly+` et `poly*ext` :

```
(define (poly- p1 p2)
  (poly+ p1 (poly*ext -1 p2)))
```

- Stratégie récursive pour la **multiplication** `(poly* p q)` :

$$(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0) \times (b_m x^m + \ldots + b_0)$$
$$= a_n x^n \times (b_m x^m + \ldots + b_0) + (a_{n-1}x^{n-1} + \ldots + a_0) \times (b_m x^m + \ldots + b_0)$$

cf TP !

```
> (poly->string (poly* p p))
"4x^10+4x^9+9x^8+4x^7-4x^6+8x^5-2x^4+12x^3+4x^2-12x+9"
> (poly->string (poly* (string->poly "2x^100-1")
                       (string->poly "2x^100+1")))
"4x^200-1"
```

- Enfin, la **valeur** d'un polynôme en un point `(poly-val p x)` :

```
> (poly-val p 1)              ; valeur en un point
6
```

cf TP !

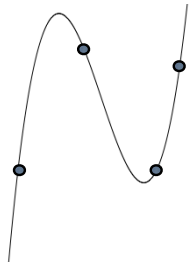
L'interpolation de LAGRANGE

- Un polynôme de degré ≤ n-1 sur un corps commutatif (ici **R**) est entièrement déterminé par ses valeurs en n points **distincts** $x_1,\ldots,x_n$

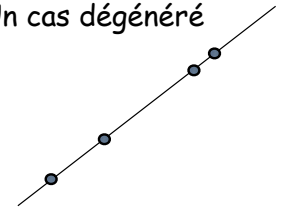
L'application : $\mathbf{R}_{n-1}[X] \xrightarrow{\sim} \mathbf{R}^n$, $P \longmapsto (P(x_1), \ldots, P(x_n))$

est un **isomorphisme** de l'e.v. des polynômes de degré ≤ n-1 sur $\mathbf{R}^n$.

Une cubique par 4 points en position générale !



Un cas dégénéré



http://fr.wikipedia.org/wiki/Interpolation_lagrangienne

- Soient donc (x_1,y_1) , … , (x_n, y_n) des points d'**abscisses distinctes**.
- Une manière effective de calculer le polynôme P tel que $P(x_i) = y_i$ pour tout i consiste à utiliser les polynômes L_i d'interpolation de Lagrange, donnés par la formule :

$$L_i(X) = \frac{(X-x_1)(X-x_2)\dots(X-x_{i-1})(X-x_{i+1})\dots(X-x_n)}{(x_i-x_1)(x_i-x_2)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_n)}$$

- Ce polynôme $L_i(x)$ a la vertu suivante : $L_i(x_j) = \delta_{i,j}$
- D'où le polynôme P cherché : $\boxed{P(X) = \sum_i y_i L_i(X)}$
- Nous aurons besoin de construire des polynômes constants a et des polynomes élémentaires X - a . A rajouter au type abstrait :

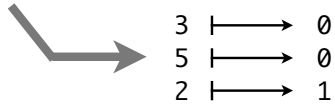
```
(define (poly-const a)        ; le polynôme constant a
  (if (zero? a) poly0 `((,a 0))))

(define (poly-rac a)          ; le polynôme élémentaire X-a
  (if (zero? a) '((1 1)) `((1 1) (,(- a) 0))))
```

- La fonction `(poly-lagrange xi Lx)` construit le polynôme L_i , où `Lx` est la liste $(x_1 \dots x_{i-1}\ x_{i+1} \dots x_n)$:

```
(define (poly-lagrange xi Lx)
  (define (iter Lx acc)
    (if (null? Lx)
        acc
        (let ((x1 (car Lx)))
          (iter (cdr Lx)
                (poly* (poly*ext (/ (- xi x1)) (poly-rac x1))
                       acc)))))
  (iter Lx (poly-const 1)))
```

$L_i(x_j) = \delta_{i,j}$

```
> (poly->string (poly-lagrange 2 '(3 5)))
"1/3x^2-8/3x+5"
```

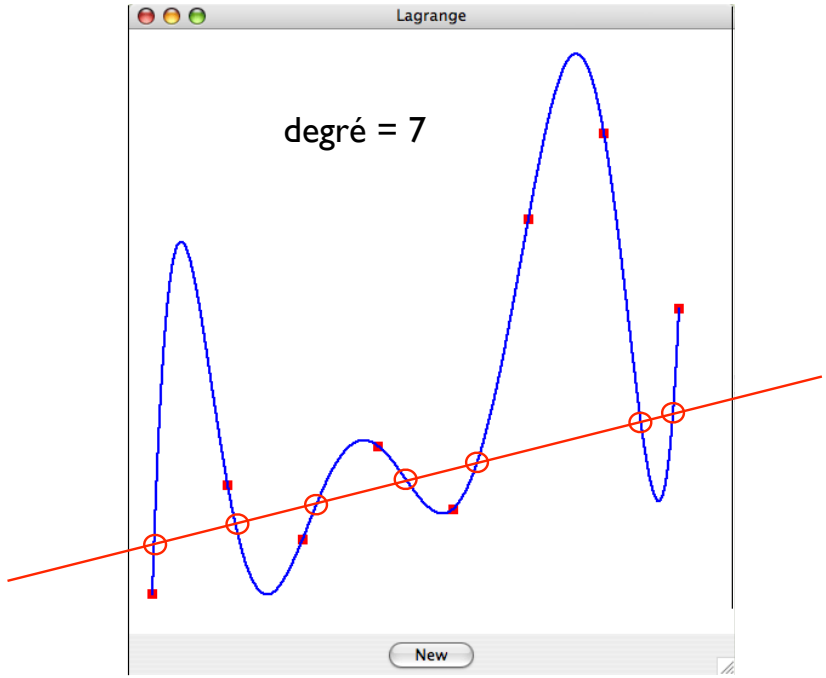


- On peut alors construire le **polynôme d'interpolation** des points (x_1,y_1) , … , (x_n, y_n) :

```
(define (interpol Lpoints)      ; Lpoints est une liste de posn
  (let ((Lx (map posn-x Lpoints)))   ; la liste des abscisses
    (define (supprimer x L)      ; on sait que x ∈ L
      ...)
    (define (iter Lpts acc)      ; j'ai besoin de l'original Lpoints
      ...)
    (iter Lpoints ...)))
```

cf TP !

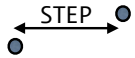
```
> (define P (interpol (list (make-posn 3 -2) (make-posn 5 0)
                            (make-posn 0 4))))
> (poly->string P)
"3/5x^2-19/5x+4"
> (map (λ (x) (poly-val P x)) '(3 5 0))     ; vérification
(-2 0 4)

> (poly->string (interpol '((3 -2) (6 -4) (-12 8))))
"-2/3x"        ; cas dégénéré : points alignés !
```



L'interface graphique

• Un bouton permet de choisir aléatoirement N points, que l'on dessine, ainsi que le polynôme d'interpolation, de degré ≤ N-1.

```
(define N 5)                          ; nombre de points
(define SIZE 500)                     ; taille du canvas
(define STEP (quotient SIZE (+ N 1)))  ; espacement en x des points
(define LPTS '?)
(define POLY '?)
```



• Génération de la liste des points du plan espacés de *dx* = STEP :

```
(define (random-points!)
  (set! LPTS (build-list N
               (lambda (i) (make-posn (* (+ i 1.0) STEP)
                                      (random SIZE)))))
  (set! POLY (interpol LPTS)))
```

• La liste des points [coordonnées 0..SIZE] sera LPTS et le polynôme d'interpolation POLY :

```
(random-points!)
```

Avec effet de bord sur LPTS et POLY...

```
> LPTS
(#(struct:posn 83.0 342) #(struct:posn 166.0 379)
 #(struct:posn 249.0 394) #(struct:posn 332.0 233)
 #(struct:posn 415.0 405))
> (poly->string POLY)
"5.820897035105816e-07x^4-0.0005280229642623335x^3
+0.1611082885759907x^2-19.199799196787154x+1100.0"
```

• Nous aurons besoin de deux crayons, un pour les lignes et un pour les points :

```
(define PEN-LINE (make-object pen% "blue" 2 'solid))
(define PEN-POINT (make-object pen% "red" 8 'solid))
```

• Les composants graphiques de l'interface :

```
(define FRAME
  (new frame% (label "Lagrange") (x 10) (y 10)
    (stretchable-width #f) (stretchable-height #f)))

(define CANVAS
  (new canvas% (parent FRAME)
    (min-width SIZE) (min-height SIZE)
    (paint-callback (lambda (c dc)
                      (draw-lagrange dc)))))

(define BUTTON-NEW
  (new button% (parent FRAME)
    (label "New") (style '(border))
    (callback (lambda (b evt)
                (random-points!)
                (send CANVAS refresh-now)))))
```

On génère un évènement on-paint qui sera attrapé par le callback du canvas.

• La fonction (draw-lagrange dc) va dessiner les points puis le polygone, par l'intermédiaire du dc.

```
(define (draw-lagrange dc)
  (define (draw-points)
    (for [(p (in-list LPTS))]
        (send dc draw-point (posn-x p) (posn-y p))))
  (define (draw-poly)
    (for ([x (in-range STEP (+ 1 (* N STEP)))])
      (send dc draw-line x (poly-val POLY x)
                           (+ x 1) (poly-val POLY (+ x 1)))))
  (send dc clear)
  (send dc set-pen PEN-POINT)
  (draw-points)
  (send dc set-pen PEN-LINE)
  (draw-poly))
```

Faire bouger les sommets à la souris !

• Nous allons appliquer la méthode du cours 6, pages 18-21. Chacun des N sommets sera autorisé à être déplacé sur une ***verticale*** !

• Lorsque la souris est en (x,y), il faut rechercher si elle est **très proche** de l'un des N points. On retourne alors ce point, ou #f.

```
(define V #f)                          ; le point courant pour la souris

(define (pt-proche x y Lpoints)        ; retourne un posn ou #f
  (if (null? Lpoints)
      #f
      (let ((xi (posn-x (car Lpoints)))
            (yi (posn-y (car Lpoints))))
        (if (< (+ (abs (- x xi)) (abs (- y yi))) 6)
            (car Lpoints)
            (pt-proche x y (cdr Lpoints))))))
```

• On définit ensuite une **sous-classe** de canvas% dans laquelle on **redéfinit** la méthode on-event :

```
(define MY-CANVAS%              ; avec gestion de la souris
  (class canvas%
    (define/override (on-event evt)
      (case (send evt get-event-type)
        ((left-down) (define x (send evt get-x))
                     (define y (send evt get-y))
                     (set! V (pt-proche x y LPTS)))
        ((left-up) (set! V #f))
        (else (when V
                ; on modifie physiquement l'un des points de LPTS !
                (set-posn-y! V (send evt get-y))
                (send this on-paint))))))
    (super-new)))

(define CANVAS (new MY-CANVAS% ........))      le reste ne change pas !
```

• *En bougeant les sommets à la souris, la liste LPTS est modifiée !*

Le problème du double buffer

• A vitesse élevée, l'animation va produire un **scintillement** [*flickering*] dû à l'effacement trop fréquent de l'écran. La solution passe par la technique du **double-buffer**. On travaille avec un *canvas* usuel, et avec un *bitmap-dc* associé à un *bitmap* invisible qui sera effacé et sur lequel le pas d'animation se fera. Ensuite on copie le bitmap vers le canvas !

```
(define BITMAP (make-object bitmap% SIZE SIZE))
(define BITMAP-DC (new bitmap-dc% (bitmap BITMAP)))   ; le buffer invisible
(define (draw-lagrange dc)
  (define (draw-points) ...)     ; dessine dans le buffer invisible BITMAP-DC
  (define (draw-poly)   ...)     ; dessine dans le buffer invisible BITMAP-DC
  (send BITMAP-DC clear)     ; on efface le buffer invisible : pas de scintillement !
  (send BITMAP-DC set-pen PEN-POINT)
  (draw-points)
  (send BITMAP-DC set-pen PEN-LINE)
  (draw-poly)
  (send dc draw-bitmap BITMAP 0 0 'solid))
                                   ; et on copie le buffer invisible dans le buffer visible !
```

Complément pour certains projets : les threads

• Un **thread** [*processus léger*] est un objet de 1ère classe qui représente un processus en train de tourner, une fonction en train de s'exécuter. Un thread peut être suspendu ou tué. **Plusieurs programmes sous forme de threads peuvent donc être exécutés en parallèle** ! Sujet difficile, pb de synchronisations, à éviter si possible !

```
(define t1 (thread (lambda ()
                     (for i from 1 to 10
                       (print 1)))))
(define t2 (thread (lambda ()
                     (for i from 1 to 10
                       (print 2)))))
```

• Exécution :

1121211212212121221 2 *par exemple...*

• Les threads sont étudiés en L3-Info...