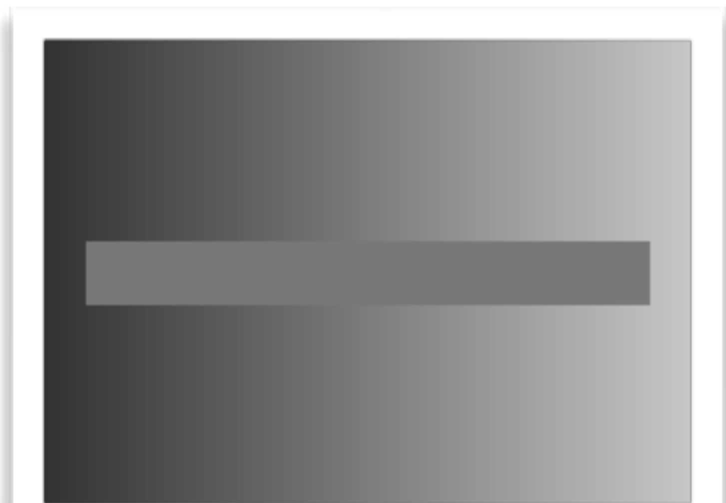
Informatique Graphique
Dessiner des lignes

Jean-Baptiste.Yunes@univ-paris-diderot.fr

9/2018

- Comment dessiner des lignes, des cercles, etc.
- La question est simple mais sa résolution n'est pas si triviale
    - surtout si l'on recherche la performance et la qualité
- Aujourd'hui on recherche le temps-réel…
    - donc minimiser le temps d'exécution des primitives est crucial
- La complexité algorithmique n'est donc pas un pur jeu de l'esprit…

- Une des difficultés est aussi d’obtenir un aspect visuel correct
- Ce n’est pas trivial en infographie…
  - Les images sont produites dans un espace discret
  - L’œil humain (et le cerveau) a des caractéristiques très surprenantes
    - Vous connaissez les illusions d’optique…



La barre du milieu est uniforme dans sa couleur… Non ? Si ! Non ! Si ?

- Aujourd’hui les dispositifs d’affichage sont discrets
  - Grille de points
- Hypothèse de départ
  - On possède des fonctions de manipulation des atomes graphiques
    - Le pixel : **pic**ture **el**ement
    - setPixel(x,y)

- Dessiner un segment entre deux points d'une grille
  - dans le monde continu il n'y a pas de difficultés, il s'agit de l'ensemble des points
    $$S_{[A,B]} = \{P_t \,/\, t \in [0,1], P_t = (1-t)A + tB\}$$
  - pour en avoir une représentation discrète «fidèle», il faudrait choisir des *pas* pour itérer sur t
    - les pas t adéquats ne sont, en général, pas des entiers par conséquent soumis à des approximations machine
      - les pas sont des rationnels, c'est ce qui nous sauvera à la fin…

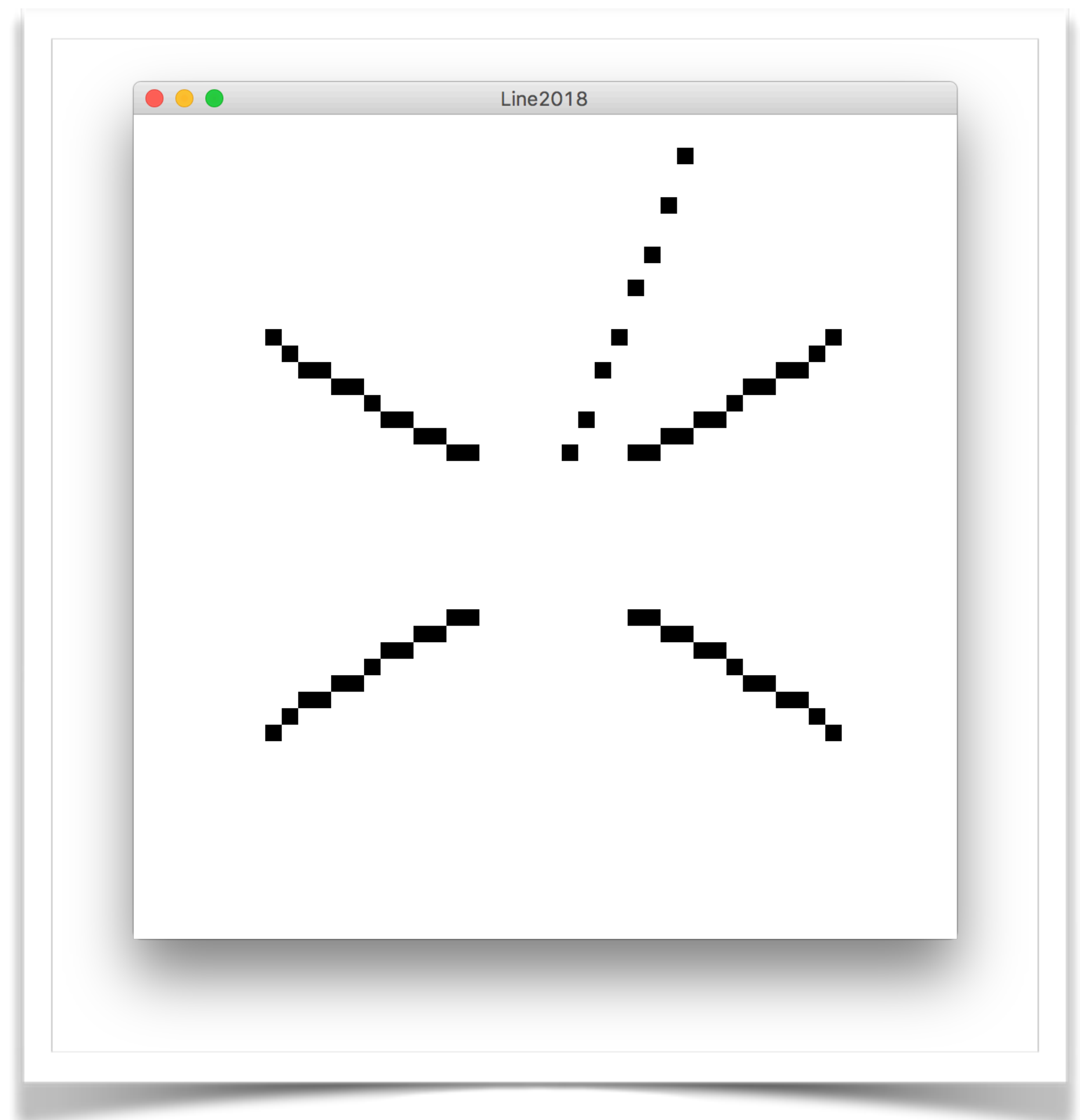
- Une autre façon de voir serait d'utiliser la représentation sous la forme de fonction
  - L'équation d'une droite est $y = ax + b$
    - a est la pente de la droite
    - Attention on ne peut représenter une verticale de cette manière!
  - Il suffit de faire varier x (on contrôle le pas horizontal) et on obtient la suite des y
    - Comment faire varier x ?

- Sans perte de généralité on peut se ramener par translation à l’obtention d’un segment dont l’une des extrémités est l’origine des coordonnées
    - [AB] c’est le segment [O,B-A] translaté par $\overrightarrow{OA}$
    - segment(A,B) -> segmentTranslate($T^{-1}_{\overrightarrow{OA}}(B)$, $T_{\overrightarrow{OA}}$)
- La droite que l’on cherche à dessiner se réduit donc à
    - $y = ax$

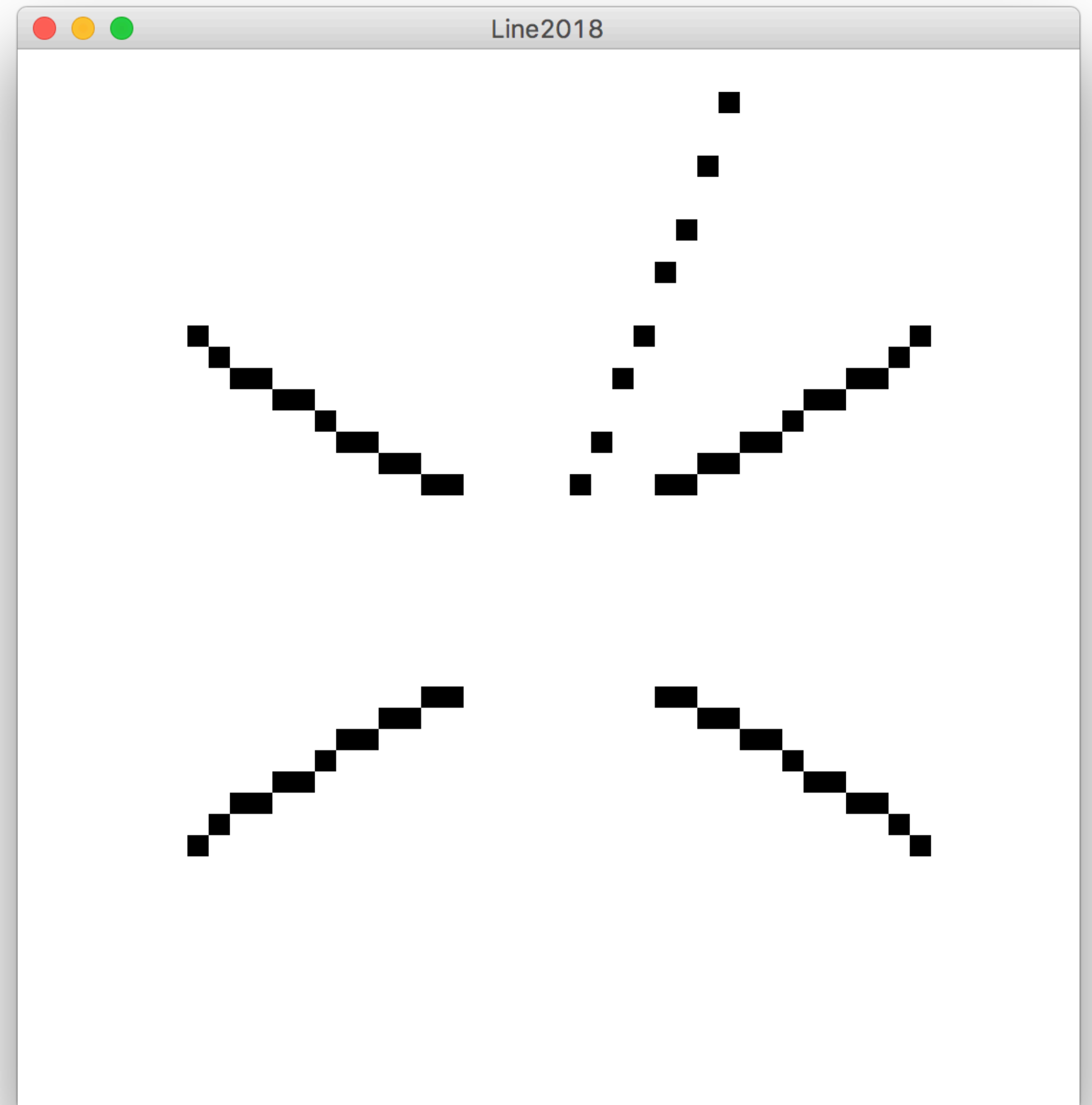
- L’extrémité du segment peut-être dans le demi-plan gauche ou droit
    - Cela influe sur la façon d’itérer (gauche à droite ou droite à gauche)
- Une simple symétrie permet de se ramener au problème dans le demi-plan droit :
    - segmentTranslate(P,f) ->
      if (P.x>=0) segmentDroit(P, f)
      else segmentDroit($S_{x=0}(P)$, $f \circ S_{x=0}$)

- segmentDroit(P,f) -> algoFonctionLineraire(P,f)

- algoFonctionLineaire(P,f) ->
  for i in [0,P.x] setPixel(f(i,P.y/P.x*i))



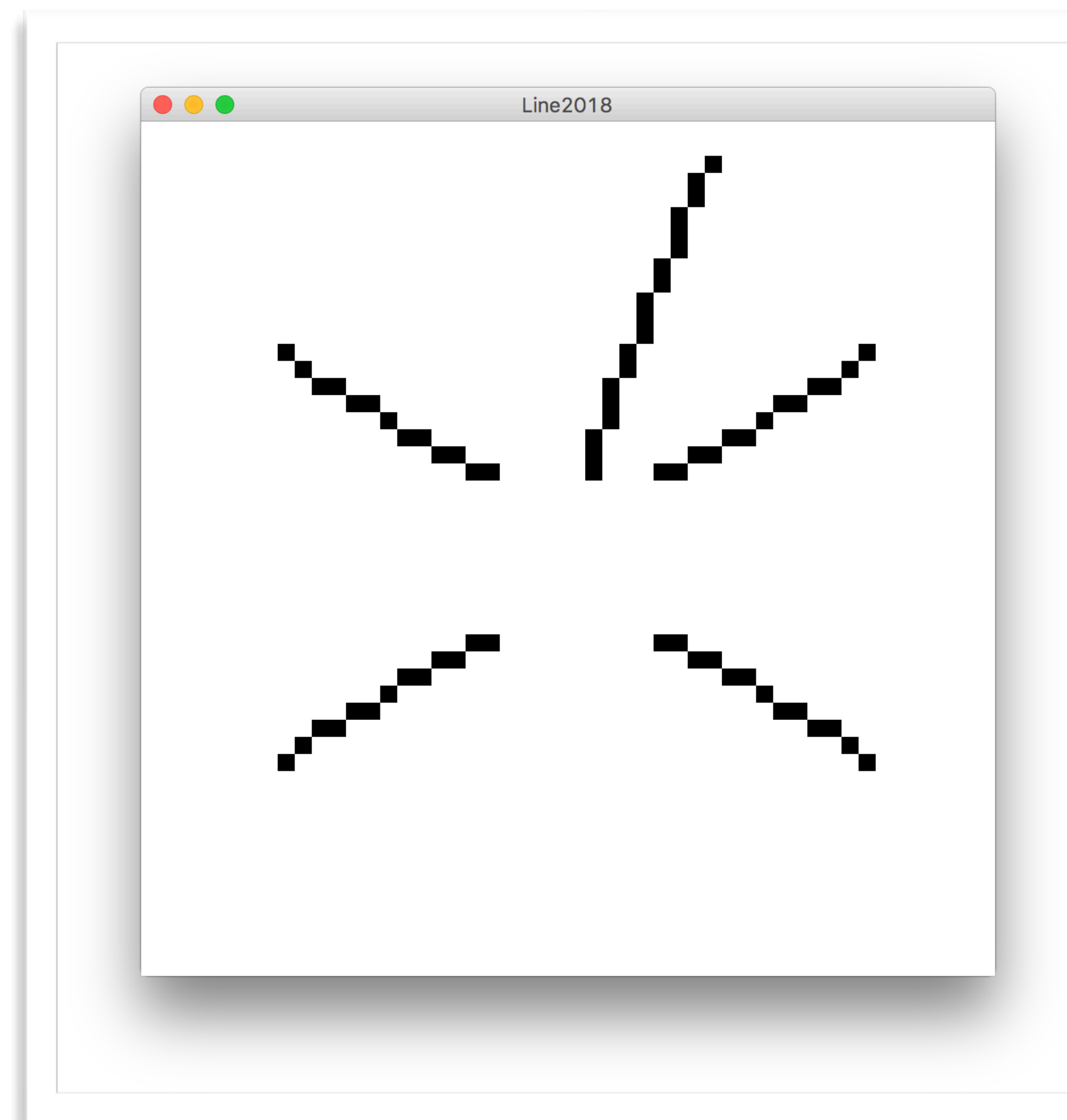
- Des problèmes
    - pas très équilibré
    - pas de symétrie
    - pas de continuité
    - ...



- La continuité
  - $y = ax$
  - la distance «verticale» entre deux points voisins «horizontalement» est
    - $y_{i+1} - y_i = a(i+1) - ai = a$
    - si $|a| > 1$, il y a discontinuité…
- Solution utiliser la symétrie diagonale!

- segmentDroit(P,f) ->
  if (abs(P.y)<P.x) segmentQuadrant(P,f)
  else segmentQuadrant($S_{y=x}(P)$, $f \circ S_{y=x}$)

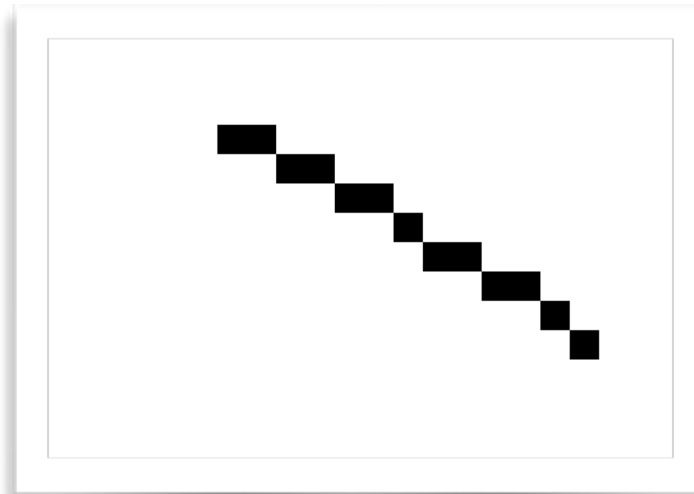
- segmentQuadrant(P,f) ->
  algoFonctionLineaire(P,f)



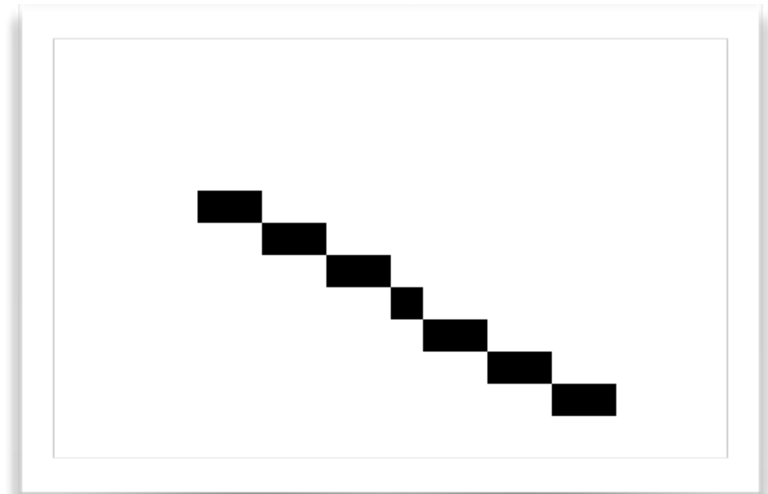
- Une optimisation ?
  - Le calcul pente*i fait appel à la multiplication flottante
  - Si l’addition flottante est plus rapide que la multiplication alors
- On peut donc transformer l’algorithme car la pente est constante et le différentiel vertical entre deux points horizontalement voisins est constant :

```
algoAccumulation(P,f) ->
y = 0
for i in [0,P.x] { setPixel(f(i,y)); y += P.y/P.x; }
```

- Problème, le résultat n'est pas identique…



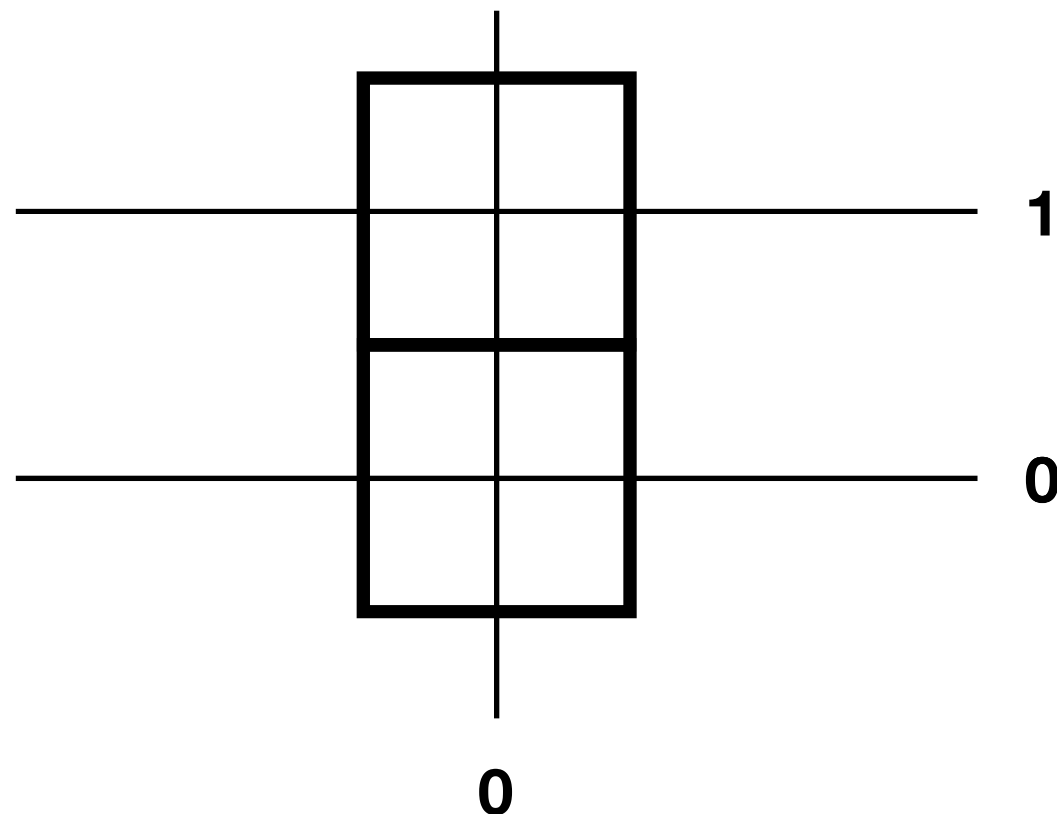
a.x
relie les points mais
n'est pas équilibré



somme_1^x a
ne relie pas les points
semble équilibré

- Dans les flottants $ax \neq \sum_{1}^{x} a$
  - C'est un sérieux problème

- Le problème de l'absence d'équilibre
  - Elle est due à l'emploi de la réduction à la partie entière
  - $\lfloor 0.9 \rfloor = 0$ alors que 0.9 est plus proche de 1 que de 0…
  - Il vaudrait mieux trouver le pixel le plus proche…

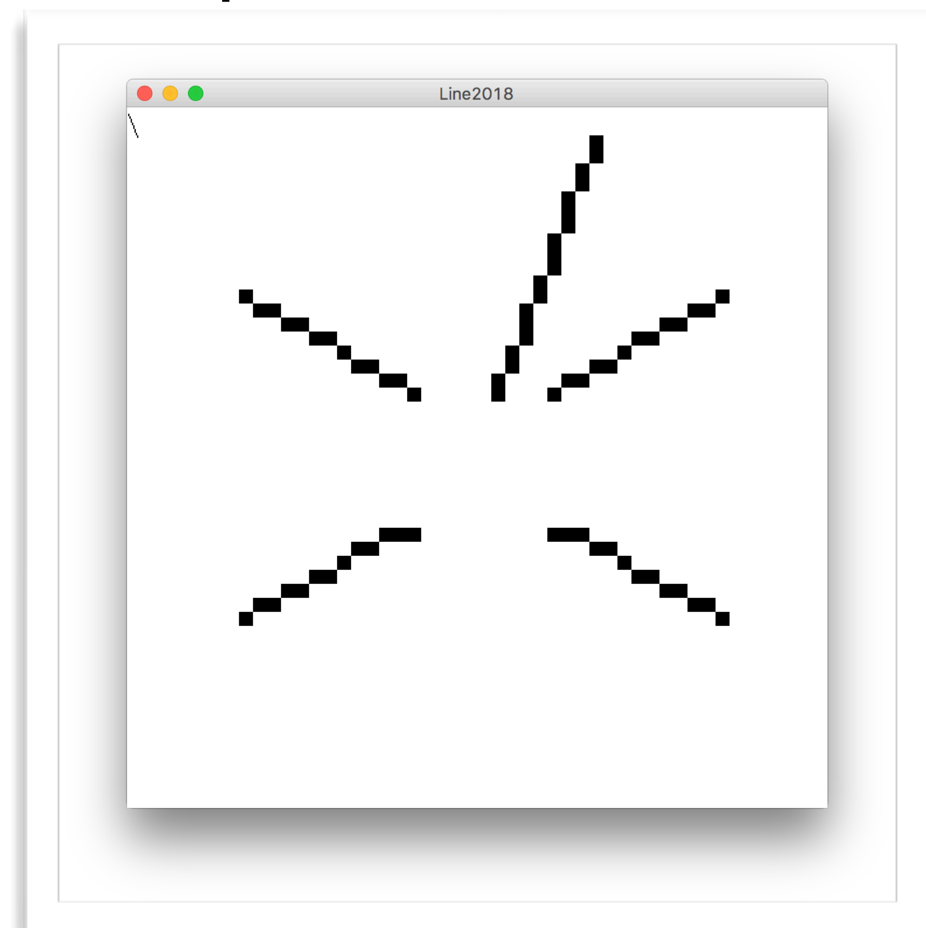


- algoAccumulationProche(P,f) ->
  y = 0
  for i in [0,P.x] { setPixel(f(i,y+0.5)); y += P.y/P.x; }

- algoAccumulationProche(P,f) ->
  for i in [0,P.x] setPixel(f(i,P.y/P.x*i+0.5));

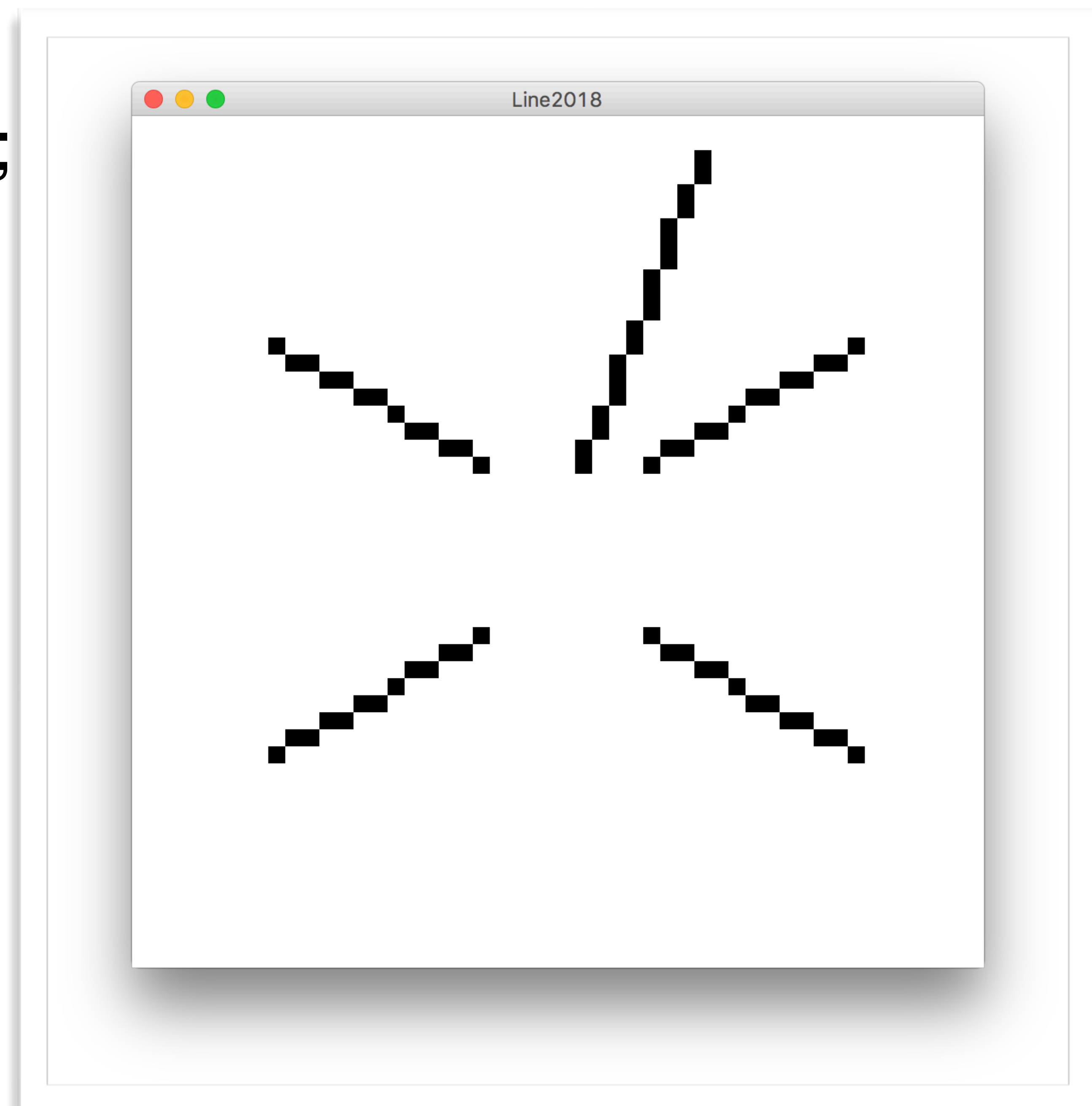
- Mais il faut régler le problème de symétrie spatiale

  - un arrondi pour les positifs
    et un pour les négatifs…



- segmentQuadrant(P,f) ->
  if (P.y<0) segmentOctant(P,f)
  else segmentOctant($S_{y=0}(P)$, $f \circ S_{y=0}$)

- segmentOctant(P,f) ->
  algoAccumulationProche(P,f);



- Est-il possible d’optimiser ?
    - On peut se passer de l’arithmétique flottante puisqu’en réalité il n’y a que des rationnels
        - Il suffit de trouver le bon coefficient multiplicateur…
        - $\frac{\Delta y}{\Delta x} i$ peut être calculé comme $i \Delta y$ dans l’espace d’échelle Δx

- algoAccumulationProche(P,f) ->
  y = 0
  for i in [0,P.x] { setPixel(f(i,y+0.5)); y += P.y/P.x; }

- En multipliant par P.x

  - algoAccumulationProcheOpt(P,f) ->
    y = 0
    for i in [0,P.x] { setPixel(f(i,y/P.x+0.5)); y += P.y; }

- En multipliant par 2

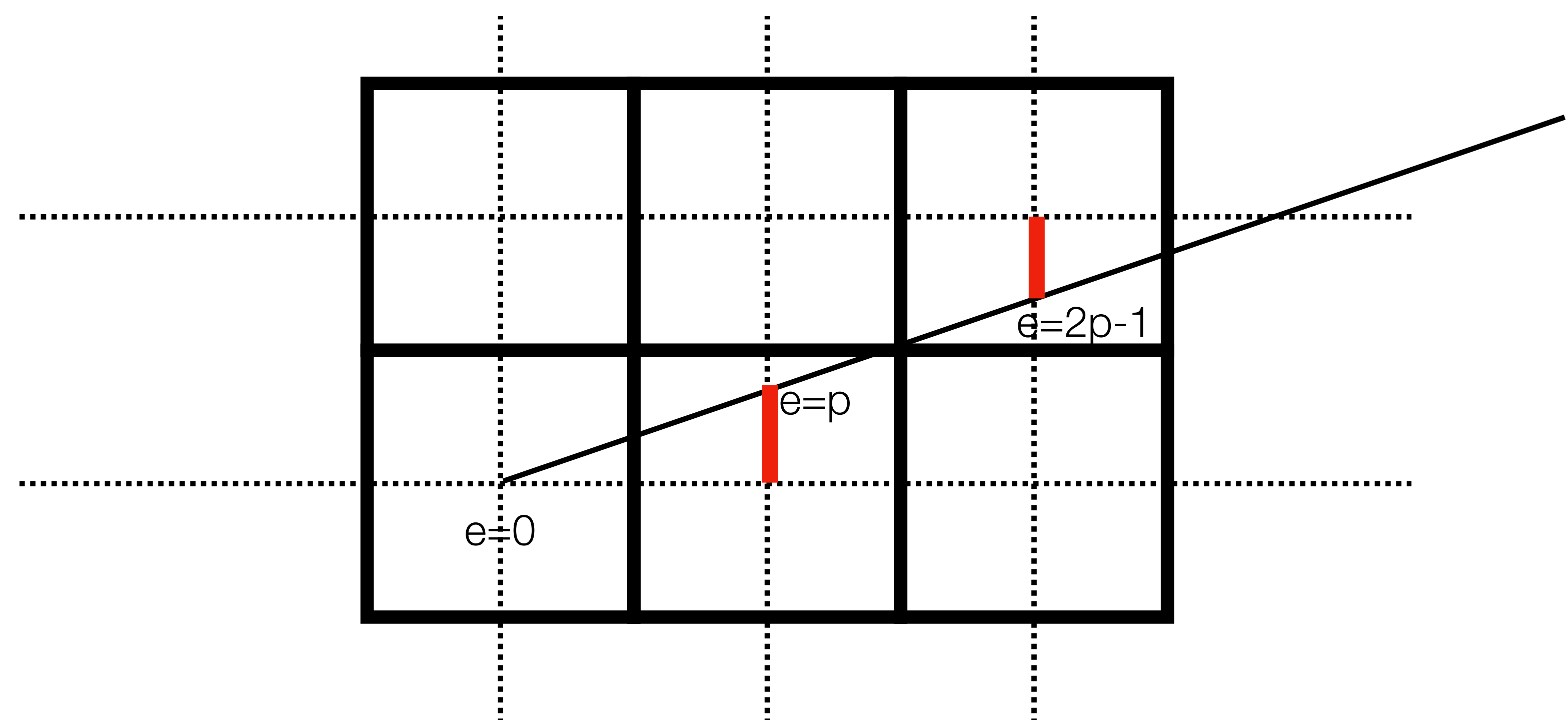
  - algoAccumulationProche(P,f) ->
    y = 0
    for i in [0,P.x] { setPixel(f(i,(y/P.x+1))/2); y += 2*P.y; }

  - algoAccumulationProche(P,f) ->
    y = 0
    for i in [0,P.x] { setPixel(f(i,(y+P.x)/2*P.x); y += 2*P.y; }

  - algoAccumulationProche(P,f) ->
    y = P.x, dx = 2*P.x, dy = 2*P.y
    for i in [0,P.x] { setPixel(f(i,y/dx); y += dy; }

- Il n’y a plus que de l’arithmétique entière…

- Pour se passer de la division (même si elle est entière) on peut alors simplement ne retenir que l'erreur…
  - à chaque itération, soit on incrémente y soit on conserve y
  - l'erreur à chaque étape est incrémentée de la pente si elle dépasse la demi-hauteur on la ramène dans l'intervalle



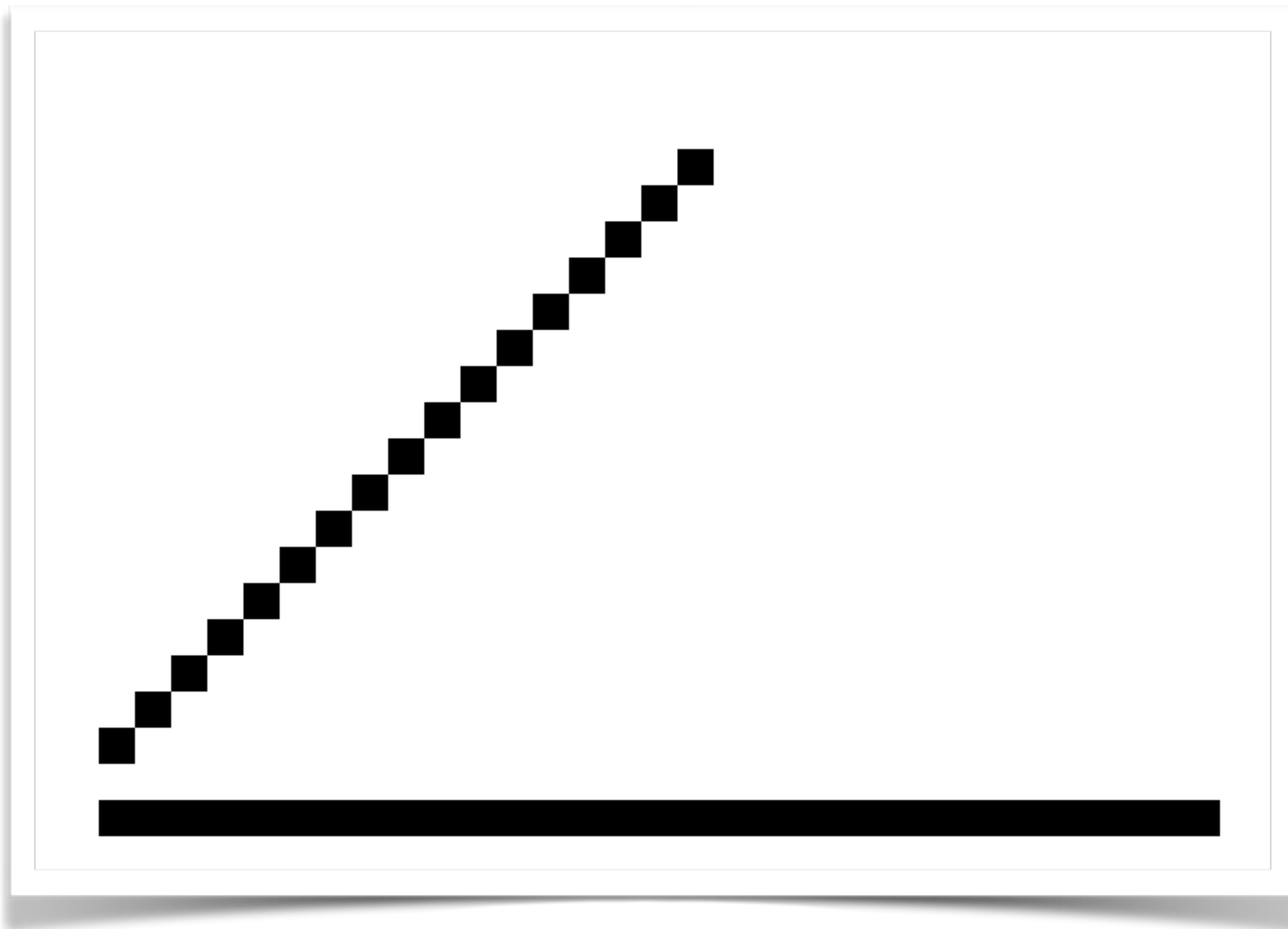
- C’est l’algorithme de Bresenham
  - ```
    algoBresenham(P,f) ->
    y = 0
    e = 0
    for i in [0,P.x] {
    setPixel(f(i,y))
    e += P.y/P.x
    if (e>0.5) { e = e-1; y++ }
    ```
- Constante entières (multiplication par 2)
  - ```
    algoBresenham(P,f) ->
    y = 0
    e = 0
    for i in [0,P.x] {
    setPixel(f(i,y))
    e += 2*P.y/P.x
    if (e>1) { e = e-2; y++ }
    ```
- Mise à l’échelle (facteur P.x)
  - ```
    algoBresenham(P,f) ->
    y = 0
    e = 0
    for i in [0,P.x] {
    setPixel(f(i,y))
    e += 2*P.y
    if (e>P.x) { e -= 2*P.x; y++ }
    ```
- Il ne reste plus que des additions/soustractions dans la boucle
    ```

- algoBresenham(P,f) ->
  ```
  y = 0
  e = 0
  for i in [0,P.x] {
  setPixel(f(i,y))
  e += 2*P.y
  if (e>P.x) { e -= 2*P.x; y++ }
  ```
- Sauriez-vous ne modifier e qu’une seule fois par tour ?
- Sauriez-vous ne tester qu’avec la constante 0 ?
- Une fois cela réalisé, c’est exactement Bresenham…

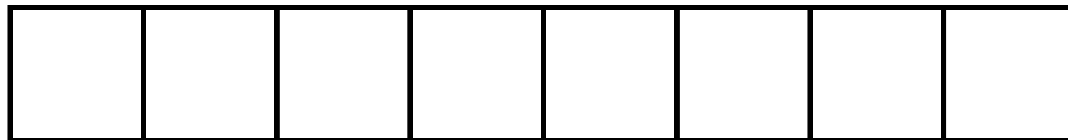
- Attention, le problème de l’optimisation de l’algorithme de tracé de ligne est très dépendant de l’architecture
  - Sur les architectures modernes, l’arithmétique flottante est quasi aussi rapide que l’arithmétique entière
  - Le problème est surtout le pipelining, donc éviter les conditionnelles
- Sauriez-vous éviter la conditionnelle dans l’algorithme précédent ?

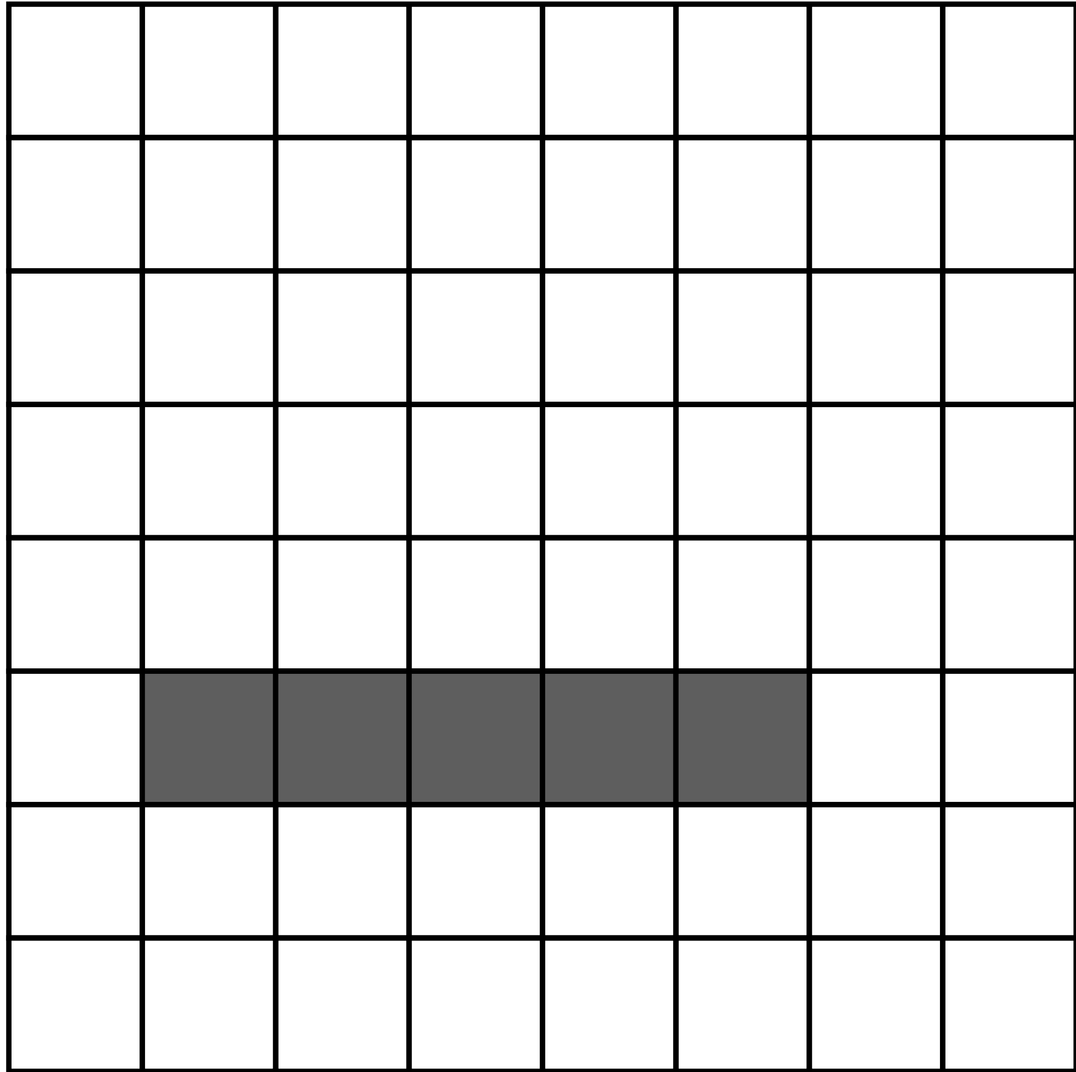
- Le problème de la symétrie
  - tracer [AB] ne donne pas le même résultat que tracer [BA]
  - C’est une erreur assez mineure (au moins visuellement)
  - Une solution est d’utiliser n’importe quel algorithme en symétrisant son comportement

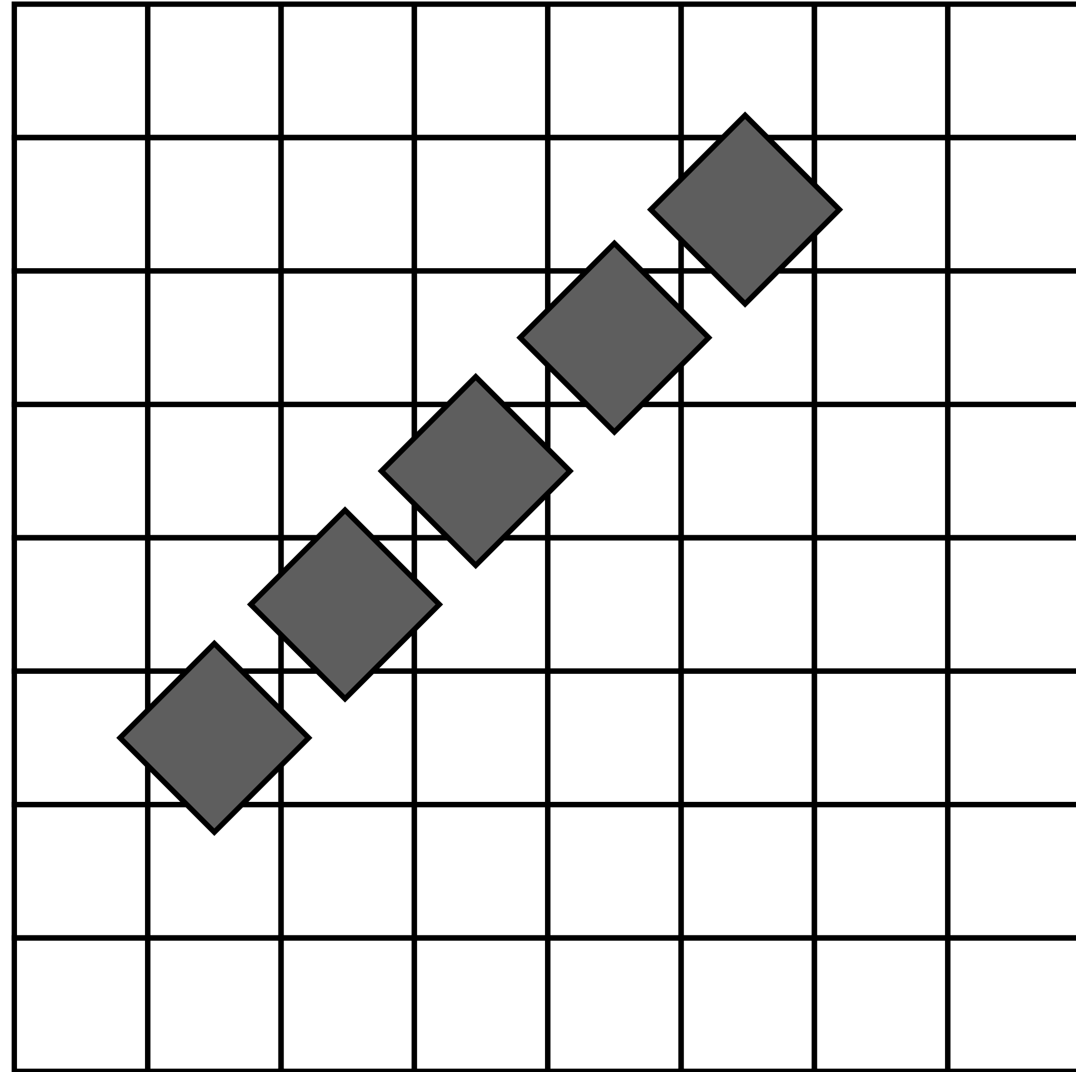
- Un problème visuel
  - Les diagonales paraissent plus claire que les horizontales/verticales



- C’est un problème de densité…
  - La diagonale (x,y)-(x+d,y+d) possède autant de pixels allumés que l’horizontale(x,y)-(x+d,y) soit d.
  - pourtant la diagonale a une longueur égale à
  - pour une droite d’une telle longueur on aurait allumé un nombre de pixels égal à $d\sqrt{2}$

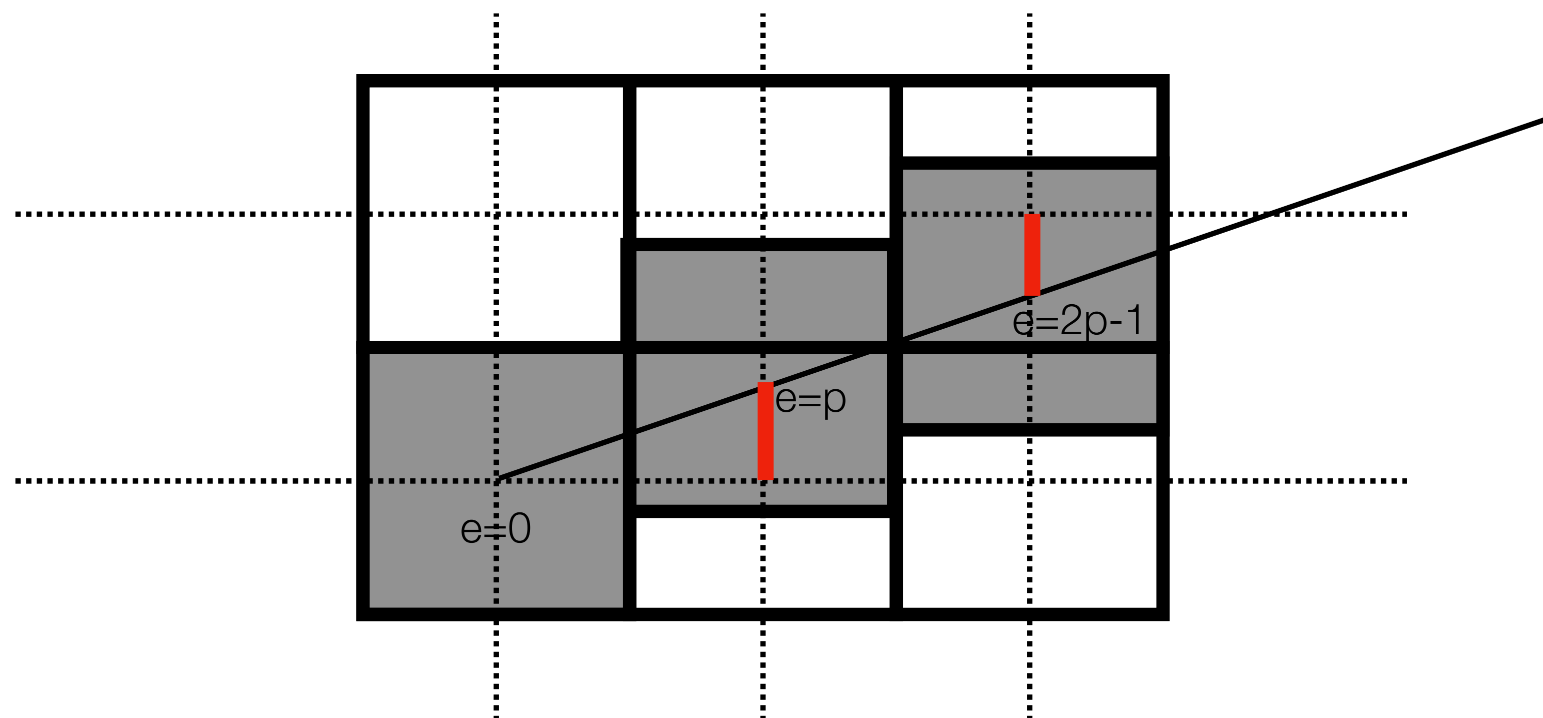


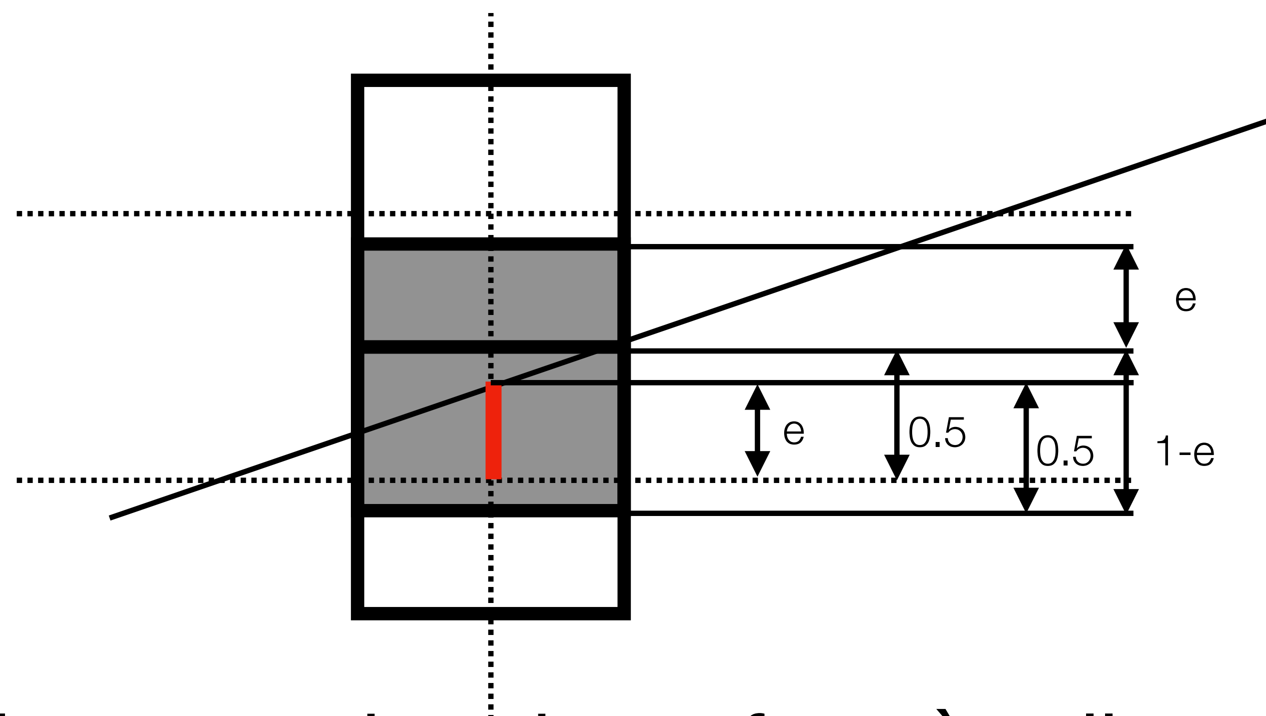




- Il faut donc remplir partiellement les pixels…

- C’est l’anti crénelage (anti-aliasing)
  - Il existe des tas de techniques
  - Une solution assez simple consiste à utiliser des niveaux de gris de sorte que la coloration soit une fonction de la surface théoriquement remplie



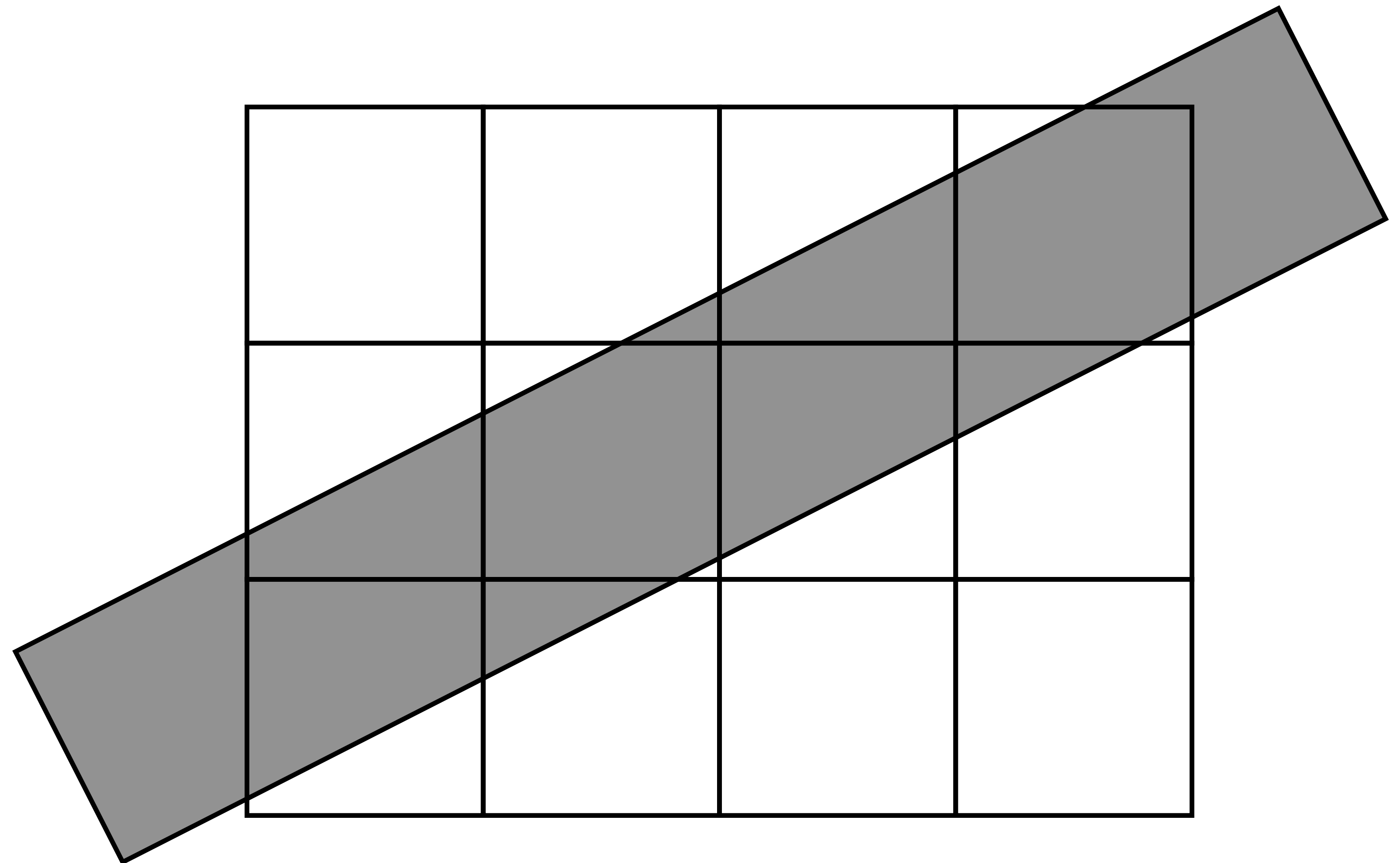


- On a donc la proportion de surface à «allumer» dans les **deux** pixels
  - si e>=0 le pixel courant e et son voisin au-dessus 1-e
  - si e<0 le pixel courant -e et son voisin du dessous 1+e
    - On peut uniformiser le calcul des valeurs : |e| et 1-|e|
- Attention il faut un cas particulier pour les diagonales

Line2018

Ops

- Une meilleure solution est l’anti-crénelage surfacique (area sampling)
    - plus complexe à mettre en œuvre et à optimiser



- Références
  - Algorithm for computer control of a digital plotter
    Jack E. Brenseham
    1965, IBM Systems Journal, Vol. 4, No. 4