Numerical Recipes in C
allocation multidimensionnelle
-
la gestion mémoire pour les PAS nulls

Lionel Lacassagne
Laboratoire d'Informatique de Paris 6
Sorbonne Université (UPMC)
Lionel.lacassagne@lip6.fr

La manière habituelle de faire #1

- Illisible, error-prone et lente, elle a tout pour plaire …
- Allocation 1D avec linéarisation des adresses:
- byte *p; p = (byte*) malloc(h*w*sizeof(byte))
- Initialisation via double boucle

```
For(i=0; i<h; i++)
  For(j=0; j<w; j++)
    P[i*w+j] = i+j;
```

- Initialisation via simple boucle (loop collapsing)

```
For(k=0; k<h*w; k++)  p[k]=0
```

La manière habituelle de faire #2

- Traitement simple: somme des points sur un voisinage 3x3 autour de (i,j)
- Y[i*w+j]=p[(i-1)*w+(j-1)] + p[(i-1)*w+(j)] + p[(i-1)*w+(j+1)] + p[i*w+(j-1)] + p[i*w+(j)] + p[i*w+(j+1)] + p[(i+1)*w+(j-1)] + p[(i+1)*w+(j)] + p[(i+1)*w+(j+1)]
- Complexité: 9 MUL + 21 ADD pour le calculs des adresses

- On développe et on réorganise
- Y[i*w+j]=p[i*w+j-w-1] + p[i*w+j-w] + p[i*w+j-w-1] + p[i*w+j-1] + p[i*w+j] + p[i*w+j+1] + p[i*w+j+w-1] + p[i*w+j+w-1] + p[i*w+j+w-1]
- Complexité: 9 MUL+22 ADD …

- Et si c'était une matrice où chaque "point" contient 3 champs ? (ex: image RGB) et on veut faire la moyenne des pixels verts (offset de 1)
- Y[i*3*w+j+1]=p[(i-1)*3*w+(j-1)+1] + p[(i-1)*3*w+(j)+1] + p[(i-1)*3*w+(j+1)+1] + p[i*3*w+(j-1)+1] + p[i*3*w+j+1] + p[i*3*w+(j+1)+1] + p[(i+1)*3*w+(j-1)+1] + p[(i+1)*3*w+j+1] + p[(i+1)*3*w+(j+1)+1]
- Complexité: 18 MUL + 30 ADD

Ce qu'on voudrait pouvoir faire

- En mono-canal
- Y[i][j]= p[i-1][j-1] + p[i-1][j] + p[i-1][j+1] + p[i][j-1] + p[i][j] + p[i][j +1] + p[i+1][j-1] + p[i+1][j] + p[i+1][j+1]
- Complexité: 12 ADD

- En couleur
- Y[i][j]= p[i-1][j-1].r + p[i-1][j].r + p[i-1][j+1].r + p[i][j-1].r + p[i][j].r + p[i][j+1].r + p[i+1][j-1].r + p[i+1][j].r + p[i+1][j+1].r

- Si c'était possible, cela se saurait …

Numerical Recipes in C

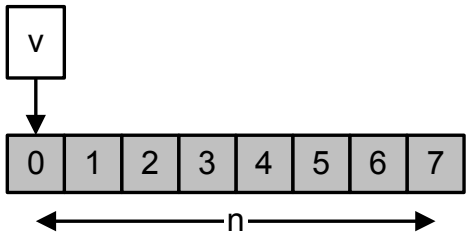
- Gestion mémoire par offset: inventée par Illife en 1961 !!!
  - John K. Iliffe (1961). "The Use of The Genie System in Numerical Calculations". *Annual Review in Automatic Programming* **2**: 25. doi: 10.1016/S0066-4138(61)80002-5.
  - But: écriture simple, lisible permettant la manipulation de tableaux à n dimensions (1D, 2D, 3D, en général, mais facilement extensible)

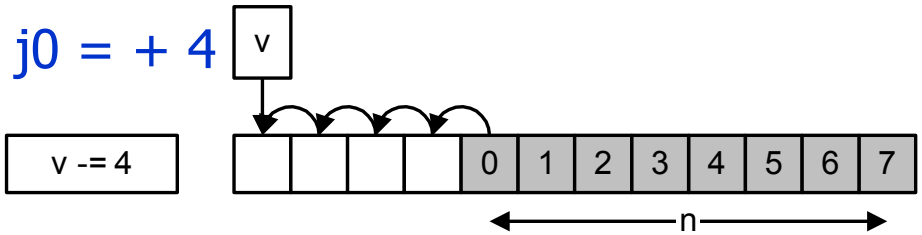
- Vecteur de Iliffe et adressage par offset
- Numerical Recipes l'a repris et l'utilise comme système de gestion mémoire pour ses bibliothèques de fonctions
  - Simplifie le passage algorithme-> langage de programmation: meme notation
  - Fortran T(i,j), Matlab T(i,j), Langage C T[i][j]

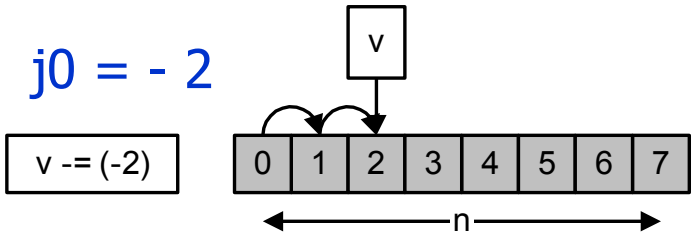
- NRC: C, C++, Fortran 77, Fortran 9x (www.nr.com)
- NRC =
  1. Interfaçage efficace avec la mémoire
  2. Boite à outils: calcul intégral, Calcul différentiel, FFT, analyse numérique

Numerical Recipes in C: allocation 1D

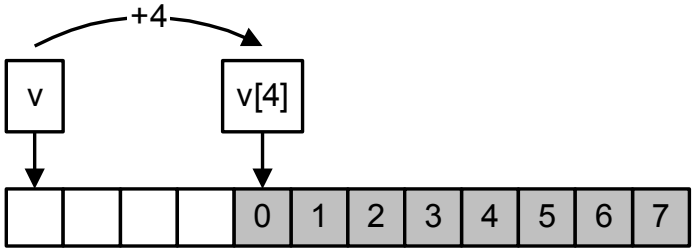
```
float* vector(int j0, int j1)
{
    n = j1 - j0 + 1;
    float* v = (float*) malloc (n * sizeof(float));
    v -= j0;
    return v;
}
```

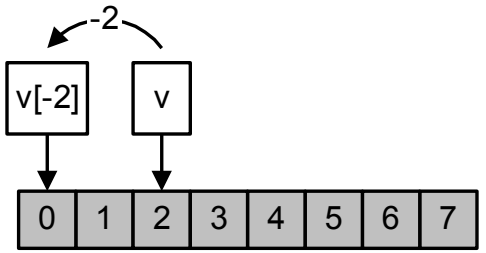






- Accès première case: v[j0] la première case véritablement allouée



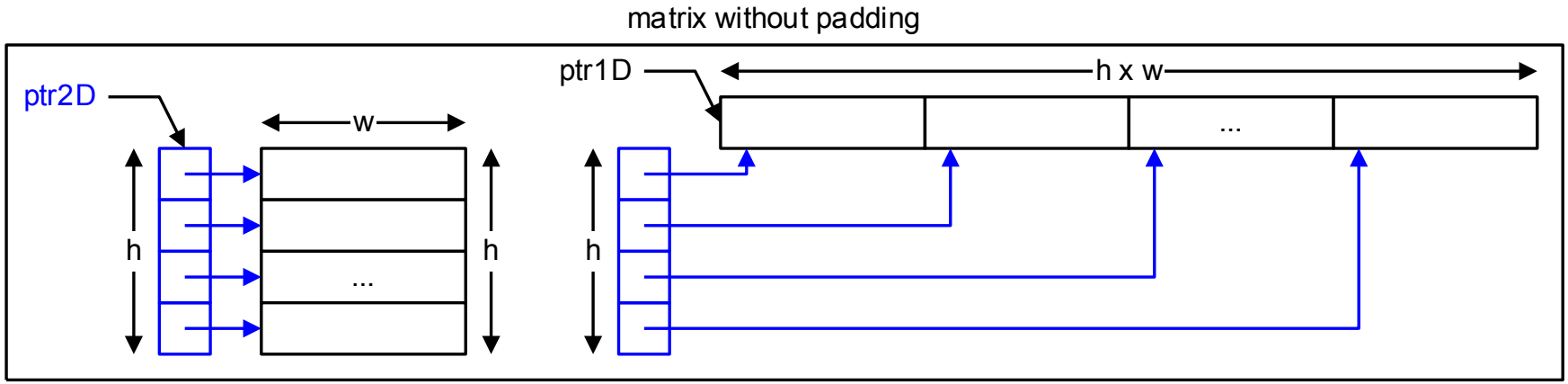


Numerical Recipes in C: allocation 1D

- Utilisation:
  - indices négatifs: vector(-10, -2)
  - indices centrés sur zéro: vector(-5, +5);
  - indices très grands (en positif ou en négatif)
    vector(1.000.000.000 , 1.000.000.007) seulement 8 cases !
    vector(-1.000.000.007 , -1.000.000.00) seulement 8 cases !
- Faire attention: on doit toujours avoir j0 <= j1
  - Ajouter un test en entrée de fonction pour le vérifier…

Numerical Recipes in C: tableau 2D

- tableau "normal"
  - Tous les points sont contigus en mémoire
  - Conversion 2D -> 1D: (i,j) -> k = i.N+j
- Wrapping
  - Accès 2D à un tableau 1D fournir par une librairie tierce, où on ne maîtrise pas l'allocation: construction uniquement de la partie "ptr2D"
  - Interfaçage avec toute bibliothèque existante (OpenCV, MIL, … matlab)



Version 2D simple

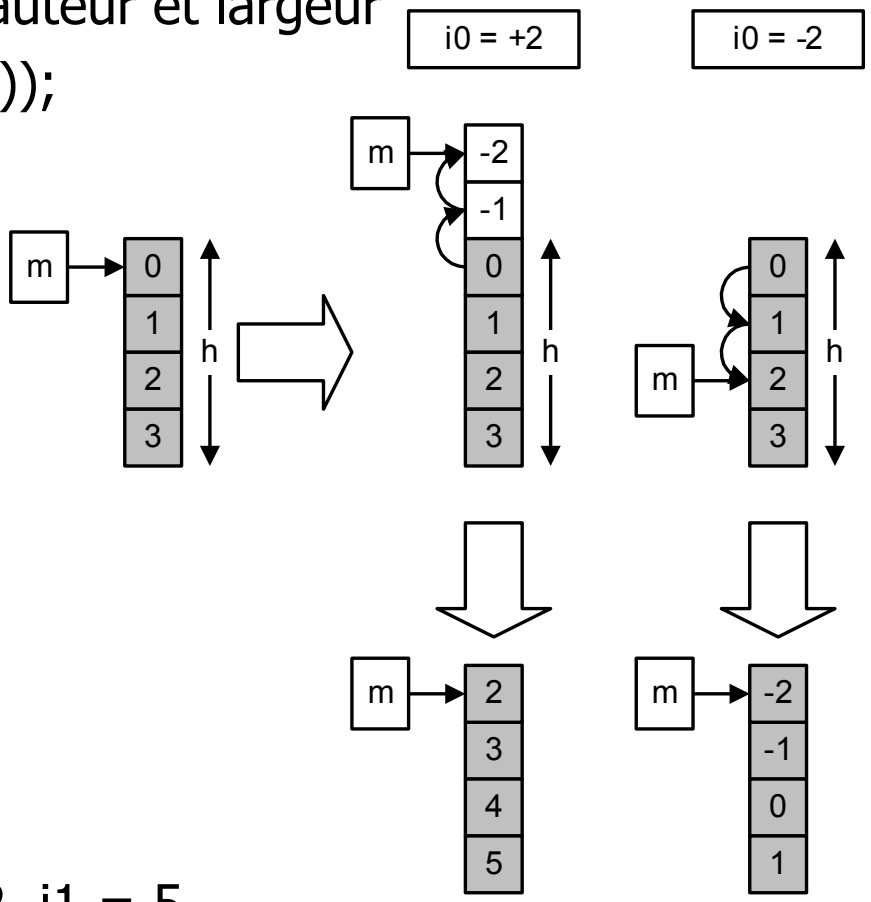
- Pour des matrices commençant à zéro: [0,h-1]x[0,w-1]
- Allouer un tableau de pointeurs vers les "débuts de lignes"
  byte **m =(byte**)malloc(h*sizeof(byte*)
- toujours faire une allocation linéaire
  m[0]=(byte*)malloc(h*w*sizeof(byte));
- Faire pointer le tableau de pointeurs au bon endroit (les débuts de lignes)
  For(i=1;i<h;i++) m[i]=m[i-1]+w;

Mais il est possible de faire beaucoup plus/mieux, la gestion par index est très polyvalente …

NRC: allocation 2D – allocation des pointeurs de lignes

```
float** matrix(int i0, int i1, int j0, int j1)
{
Float **m;
h=i1 - i0 + 1; w = j1 – j0 + 1;  // hauteur et largeur
m = (float**) malloc(h * sizeof(float));

m -= i0; // offset vertical
...
```



- Deux cas:
  - i0 = +2, i1 = 5 => h = 4
  - i0 = -2, i1 = 1 => h = 4

- Dans la suite on supposera i0 = 2, i1 = 5

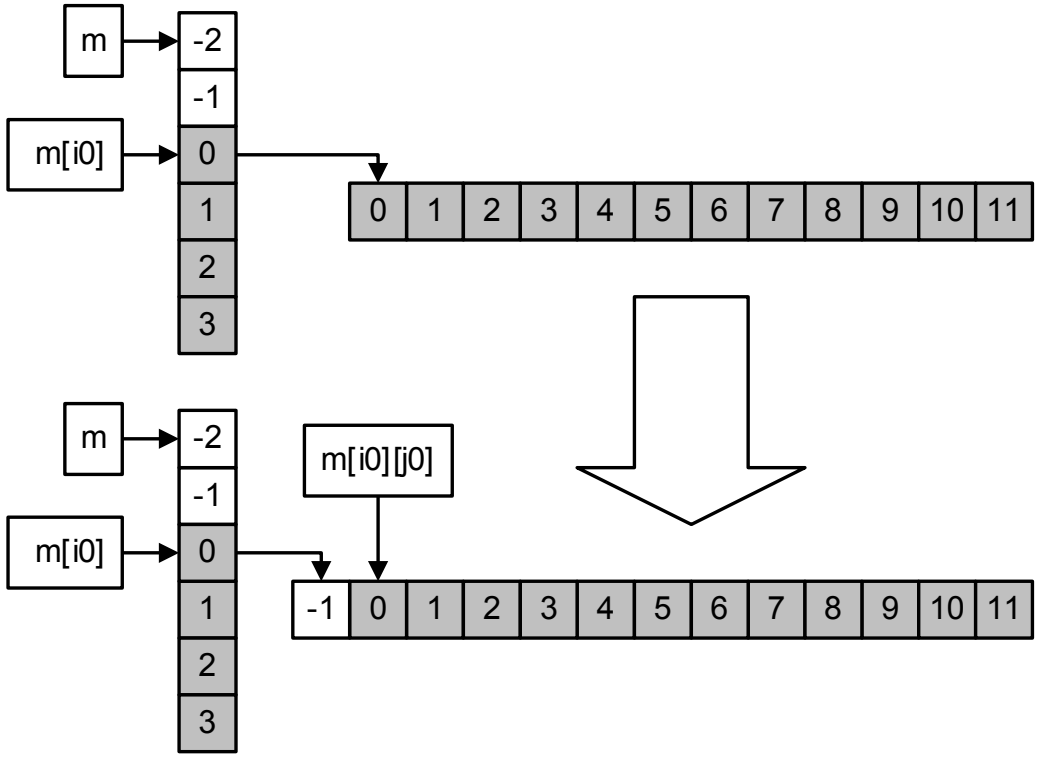
NRC: allocation 2D – allocation de la zone mémoire

m[i0] = (float*) malloc (h*w * sizeof(float));

// allocation de tout le tableau d'un seul morceau

m[i0] -= j0;

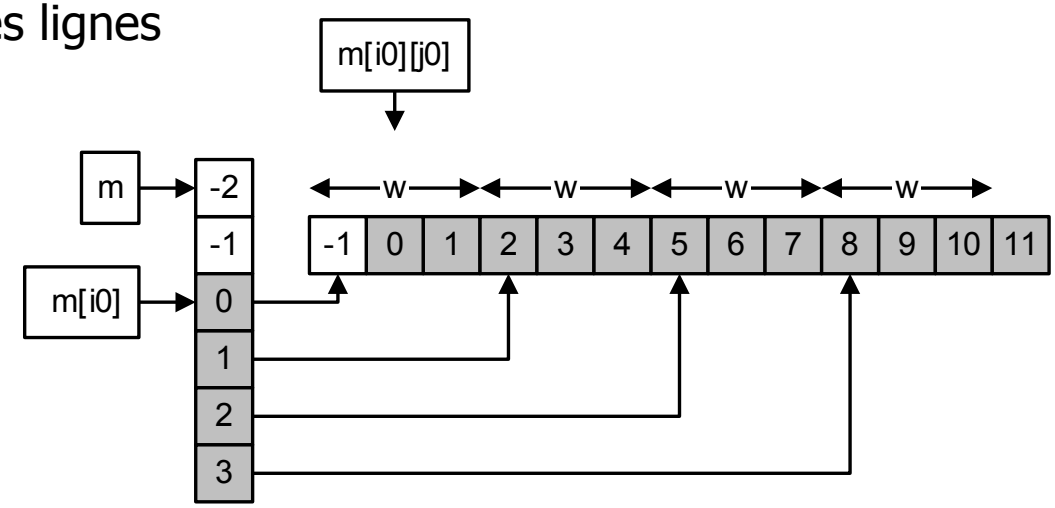
avec j0=1, j1=3 => w = 3



NRC: allocation 2D – connexion des pointeurs de début ligne

```
for(i=i0+1; i<=i1; i++)   m[i] = m[i-1] + w;
```

- Le pointeur de la première ligne pointe vers le début de la première ligne
- Les pointeurs suivant sont équidistant de la largeur d'une ligne, ils pointent donc vers le début des autres lignes



- Le code suivant initialise le tableau en gris:

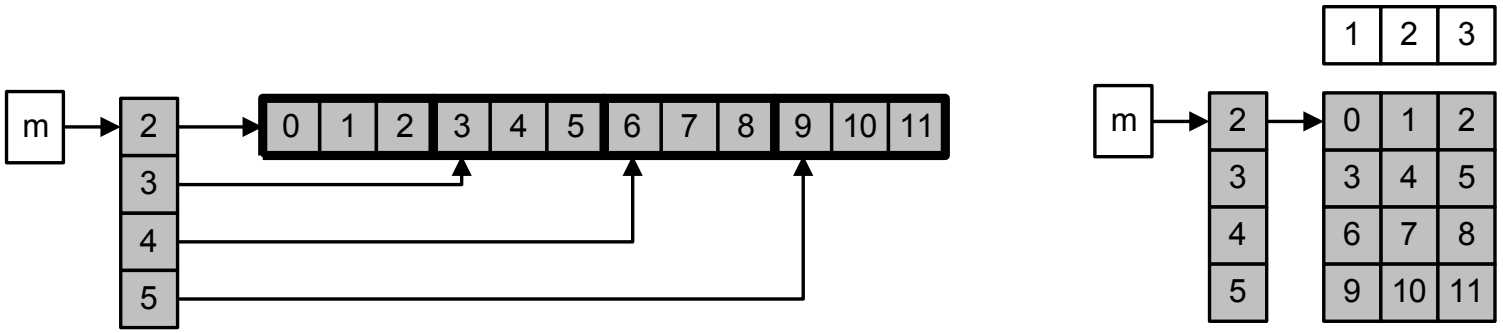
```
k = 0;
for(i=i0; i<=i1; i++)
  for(j=j0; j<=j1; j++)
    m[i][j] = k++;
```

NRC: allocation 2D – interprétation

- Les pointeurs de début de ligne permettent de "voir" la mémoire 1D comme de la mémoire 2D

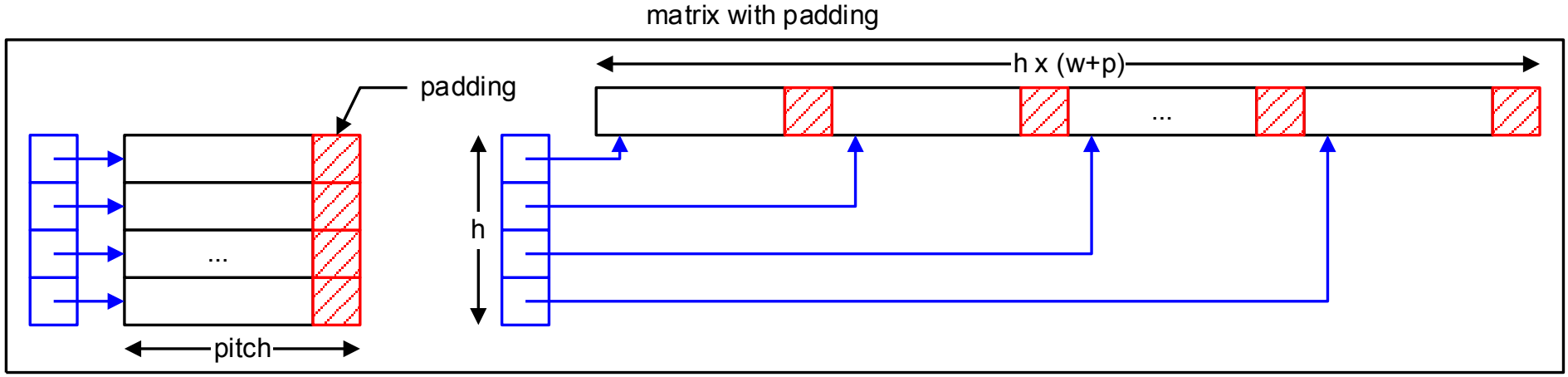
- L'adressage par offset permet de faire virtuellement commencer la numérotation verticale et la numérotation horizontale des cases à des valeurs non nulles, soit négatives, soit positives

- Comme si on avait:



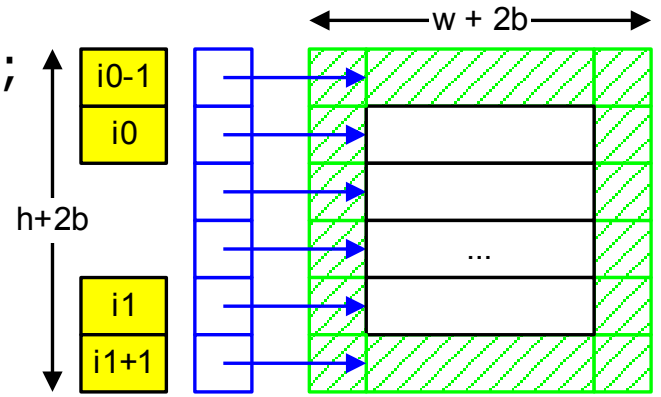
Numerical Recipes in C: image 2D avec *padding*

- image réaliste : avec *padding*
  - image BMP couleur 24 bits : début de ligne à adresse multiple de 4 pour accélérer chargement mémoire (format DIB en mémoire)
    => toutes les images sous windows sont paddées …
  - image 32 bits : toujours alignée sur multiple de 4



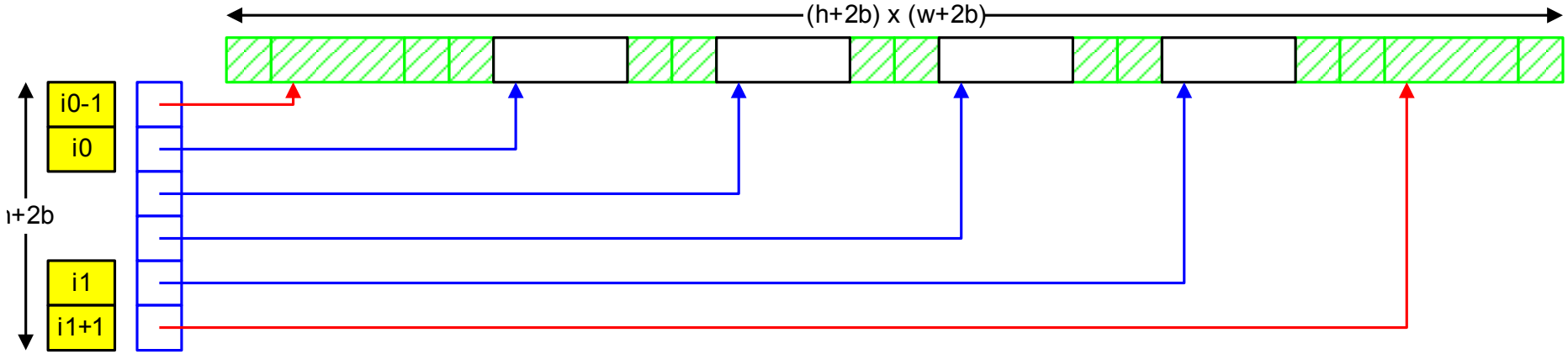
Numerical Recipes in C: matrice avec bords

- image avec bords
  - Évite de faire un traitement particuliers pour les bords (convolutions, stencil)
  - Matrice sans bord: matrix(i0, i1, j0, j1)
  - Matrice avec bord: matrix(i0-b, i1+b, j0-b, j1+b);
  - numérotation à zéro, matrix de taille n*n
    matrix(0-b, n-1+b, 0-b, n-1+b);



- intérêt NRC:

  Quelque soit le type des données, avec/sans bord: adressage identique: T[i][j]



Numerical Recipes in C: matrice avec bords

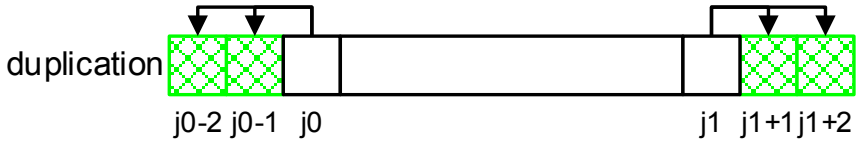
- Exemple applicatif:
  - somme sur un voisinage k x k, k=2b+1, k=3 => b=1
  - Source X=matrix(i0-b, i1+b, j0-b, j1+b);
  - Destination Y=matrix(i0, i1, j0, j1);

```
for(i=i0; i<=i1; i++) {
  for(j=j0; j<=j1; j++) {
    a0=X[i-1][j-1]; b0=X[i-1][j]; c0=X[i-1][j+1];
    a1=X[i  ][j-1]; b1=X[i  ][j]; c1=X[i  ][j+1];
    a2=X[i+1][j-1]; b2=X[i+1][j]; c2=X[i+1][j+1];
    s=a0+a1+a2+b0+b1+b2+c0+c1+c2;
    Y[i][j]=s;
  }
}
```

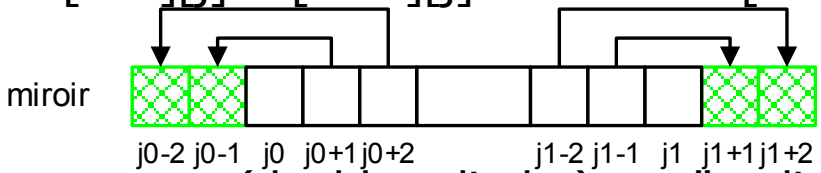
- intérêt il n'y a plus à écrire de cas particulier pour calculer la somme
  - Sur les 4 bords,
  - Sur les 4 cas coins,
  - Sinon 8 cas particuliers à gérer, en plus du cas général => risque d'erreur

NRC: initialisation des bords

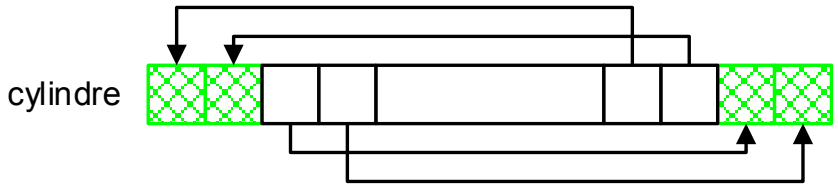
- La mémoire a été allouée, mais pas initialisée (sur les bords, ici b=2)
- Plusieurs cas possibles, en fonction du domaine applicatif:
  - Initialisation des bords à zéro
  - Duplication des bords: les bords "intérieurs" sont recopiés dans les bords "extérieurs"
    bord gauche: T[i][j0-k]=T[i][j0] bord droit T[i][j1+k]=T[i][j1]
    bord haut: T[i0-k][j]=T[i0][j] bord droit T[i1+k][j]=T[i1][j]



  - Initialisation en miroir: en symétrique par rapport au bord
    bord gauche: T[i][j0-k]=T[i][j0+k] bord droit T[i][j1+k]=T[i][j1-k]
    bord haut: T[i0-k][j]=T[i0+k][j] bord droit T[i1+k][j]=T[i1-k][j]



  - Initialisation en tore (double cylindre): on "replie la matrice sur elle-même
    bord gauche: T[i][j0-k]=T[i][j1-k] bord droit T[i][j1+k]=T[i][j0+k]
    bord haut: T[i0-k][j]=T[i1-k][j] bord droit T[i1+k][j]=T[i0+k][j]

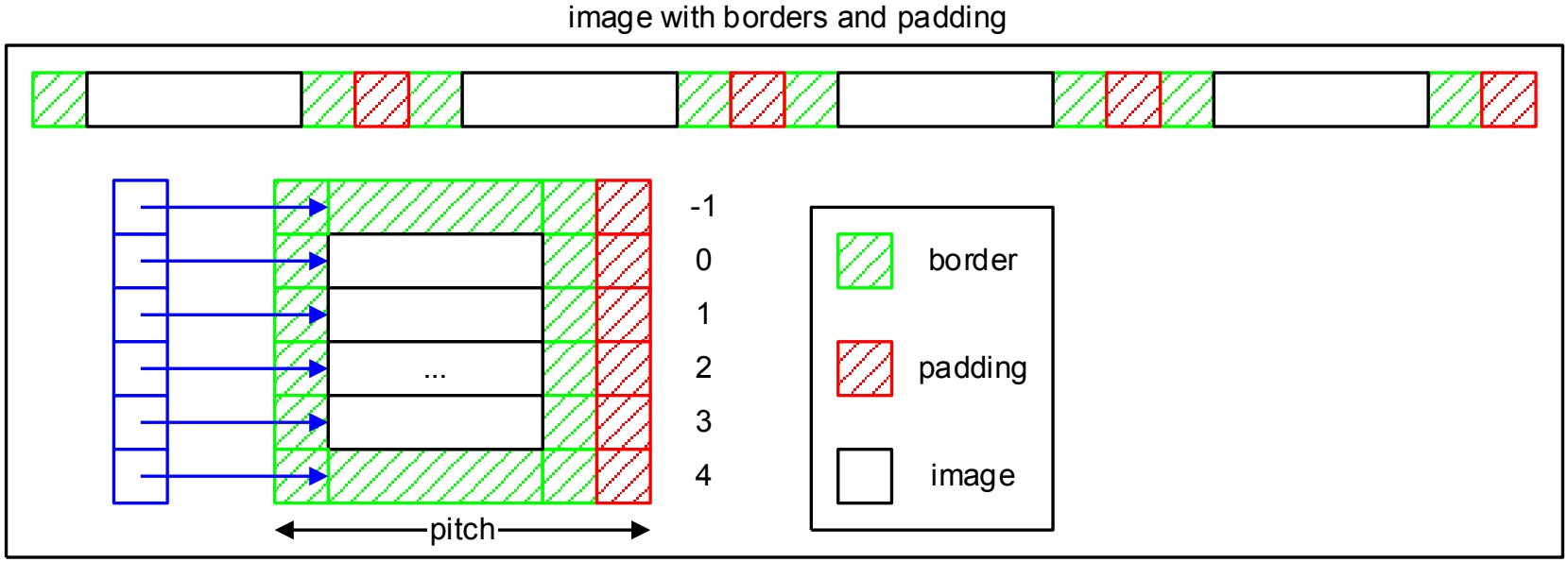


Numerical Recipes in C: image 2D

- image avec bords
  - Évite de faire un traitement particuliers pour les bords (convolutions)
  - Matrice sans bord: matrix(i0, i1, j0, j1)
  - Matrice avec bord: matrix(i0-b, i1+b, j0-b, j1+b);
  - Matrice avec bord: matrix(i0-b, i1+b, j0-b, j1+b+p);
- intérêt NRC:

  Quelque soit le type des données (avec/sans) (bord/padding)

  adressage identique: T[i][j]



Numerical Recipes in C: image 2D

- Image couleur
  - Conversion 2D -> 1D: (i,j) -> k = 3 i.N+ 3.j = 3(i.N+j)
    la linéarisation d'adresse est error prone !!!
- Avec NRC:
  - définition d'une structure = type complexe possible

  - Pixel RGB 8 bits: typedef struct {uint8 r; uint8 g; uint8 b;} **rgb8**;
  - Adressage: T[i][j].r, T[i][j].g T[i][j].b

  - Nombre complexe 32 bits: typedef struct {float32 re; float32 im;}complex32;
  - Adressage: T[i][j].re, T[i][j].im

- La mise au point et le débug sont simplifiés
- Lisibilité est accrue, la maintenance est facilitée

- De plus, il est toujour possible de repasser en 1D
  uint8 *Ti = T[i]; Ti[j]; for(j=j0; j<=j1; j++) Ti[j]

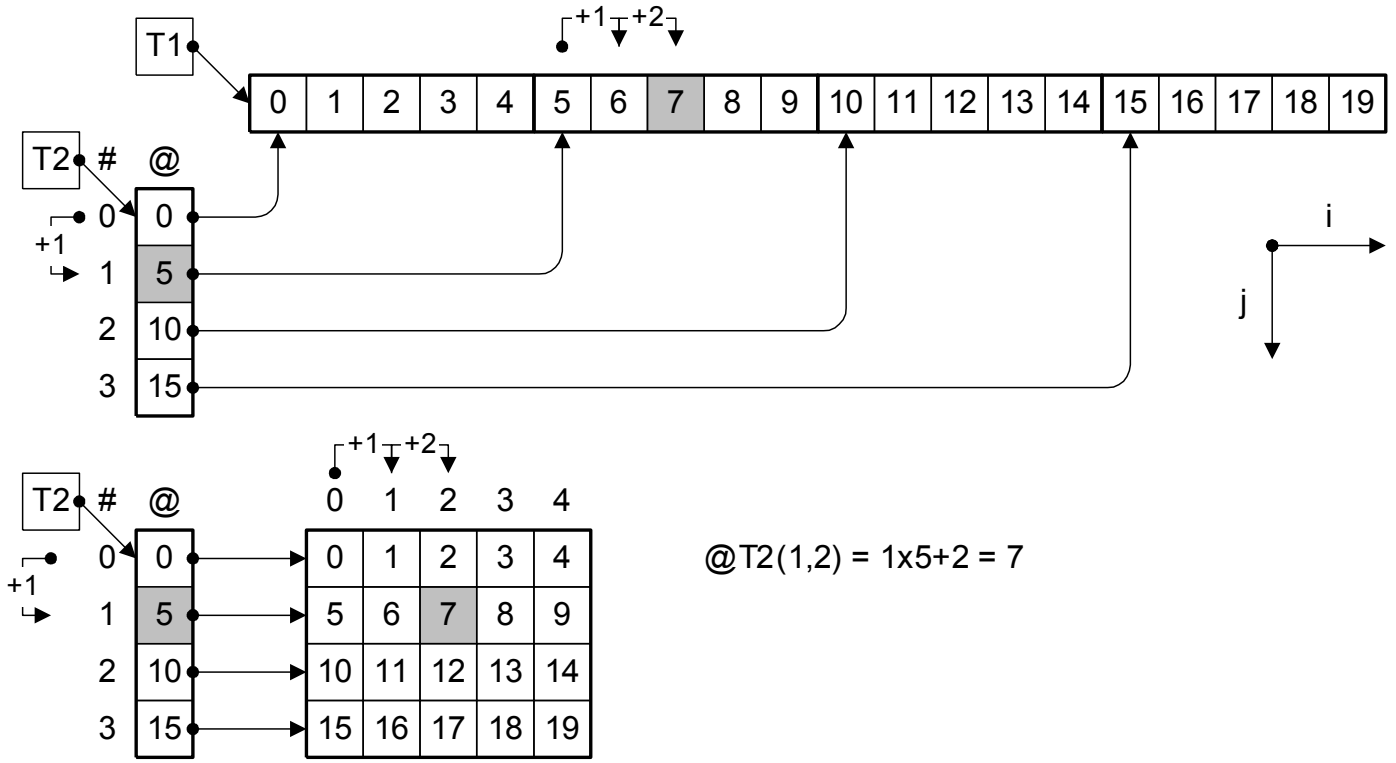
NRC : arithmétique des pointeurs 2D

- Adresssage 2D classique: transformation 2D ? 1D

  2D T[i][j] = *T(ixN+j): 1 MUL + 1 ADD = 2 OPs

- Adressage par offsets:

  offset ligne + offset colonne : 2 ADD



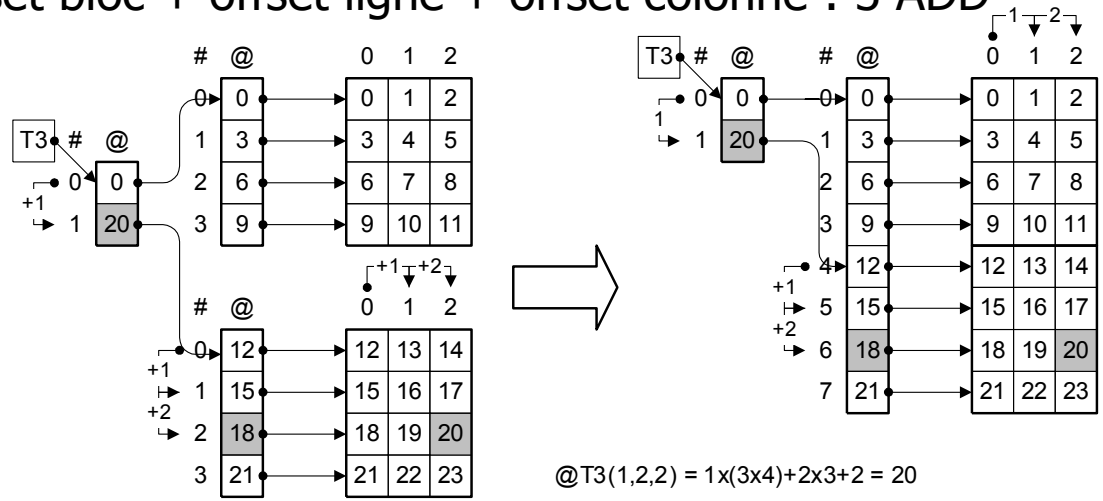
NRC : arithmétique des pointeurs 3D

- Adresssage 3D classique: transformation 23⍰1D

  2D T[k][i][j] = *T(kxN²+ixN+j): 3 MUL + 2 ADD = 5 OPs

- Adressage par offsets:

  Offset bloc + offset ligne + offset colonne : 3 ADD



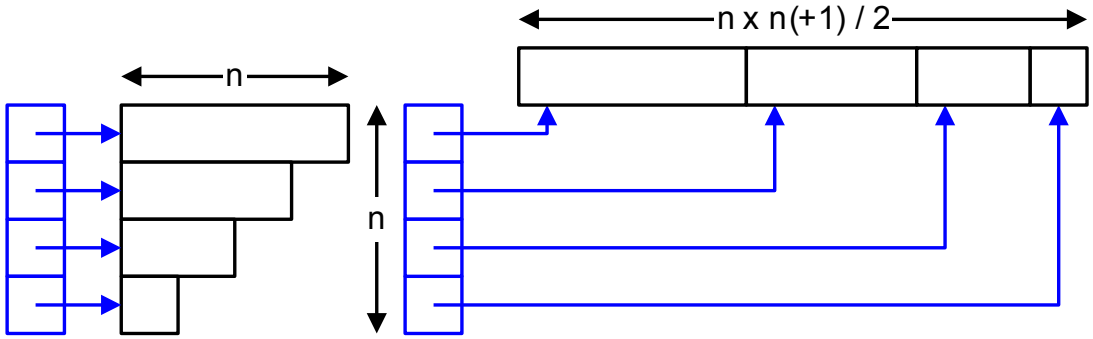
Accès mémoires

- Impact langage & type allocation
  - Fortran 77: allocation statique = le compilateur connaît l'adresse T(i,j)
    pas de pointeur, donc pas d'aliasing
  - Langage C: allocation dynamique = le compilateur ne connaît pas l'adresse
    aliasing de pointeur possible

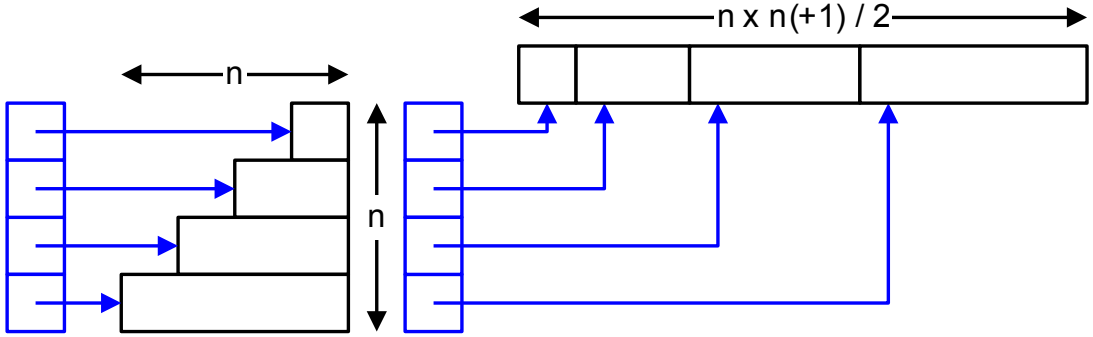
- Impact
  - F77: extraction du parallélisme facile
  - C : risque de dépendance de données car 2 pointeurs peuvent pointer la même zone mémoire ? perte de performance

Variations: matrices triangulaires

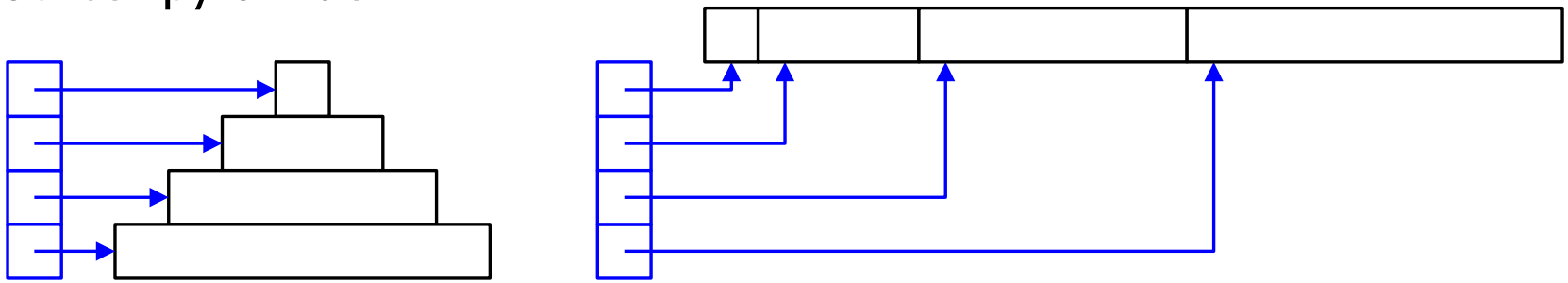
- Matrice triangulaire sup



- Matrice triangulaire inf



- Matrice "pyramide



- Remarque: ajout de bords et de padding possible
- Possibilités quasi illimitées: tout ce qui peut s'exprimer sous forme d'offset

Matrices avec bords, mais sans bord

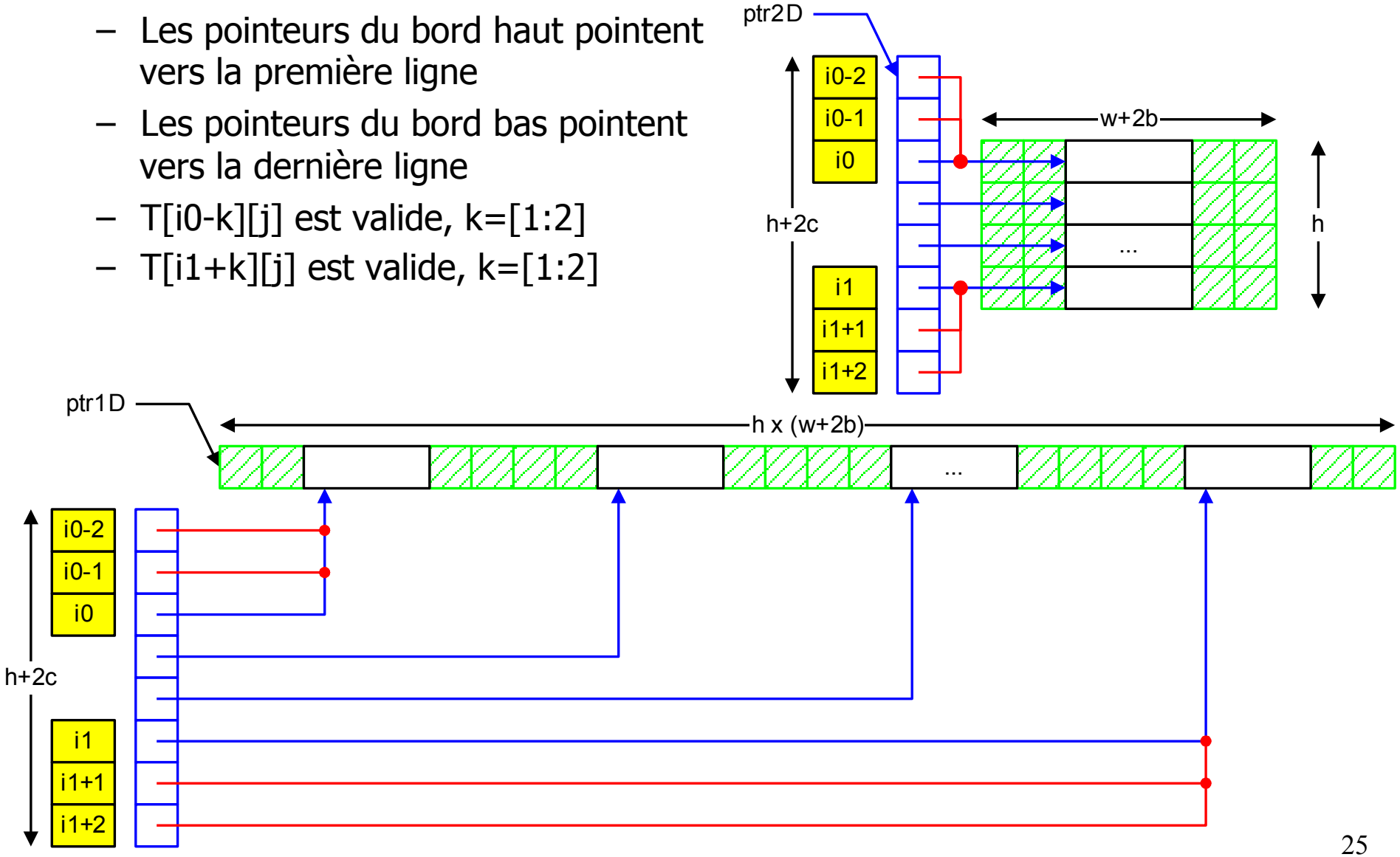
- Peut-in éviter:
  - d'allouer de mémoire supplémentaire ?
  - de passer du temps à copier les valeurs sur les bords
- D'avoir des bords, mais sans les allouer et sans y copier des valeurs

- Oui pour les bords haut et bas
  - grâce à des manipulations de pointeurs

- Non pour les bords gauche et droit
  - Si les bords ne sont pas alloués, gestions des cas particuliers par du code supplémentaire

Matrice avec clamping

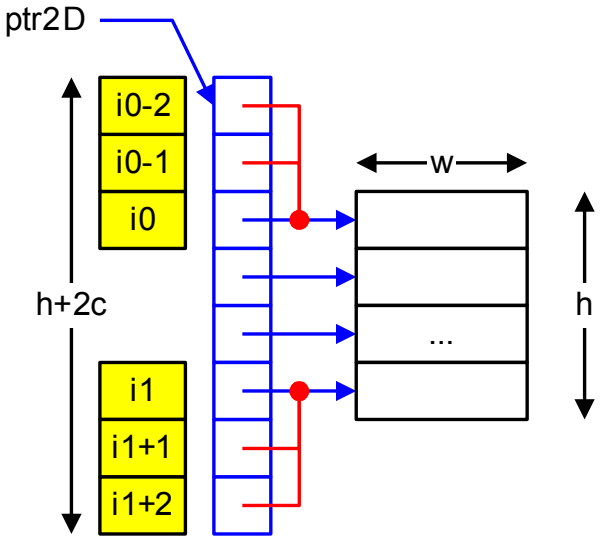
- Matrice avec bord horizontal de 2 et clamping vertical de 2
  - Les pointeurs du bord haut pointent vers la première ligne
  - Les pointeurs du bord bas pointent vers la dernière ligne
  - T[i0-k][j] est valide, k=[1:2]
  - T[i1+k][j] est valide, k=[1:2]

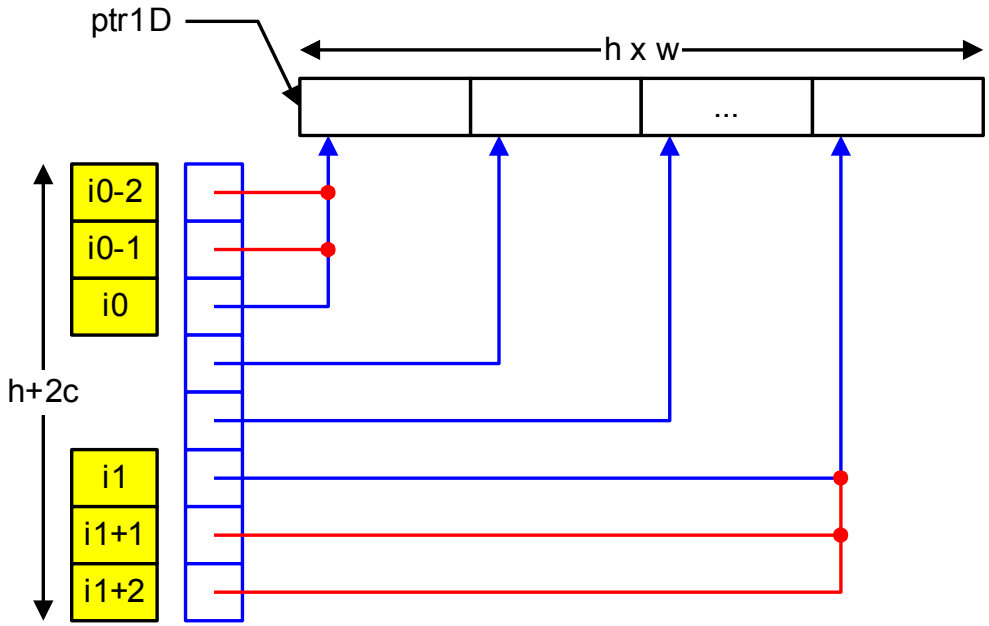


Matrice avec clamping des bords verticaux

- Schéma avec c=2

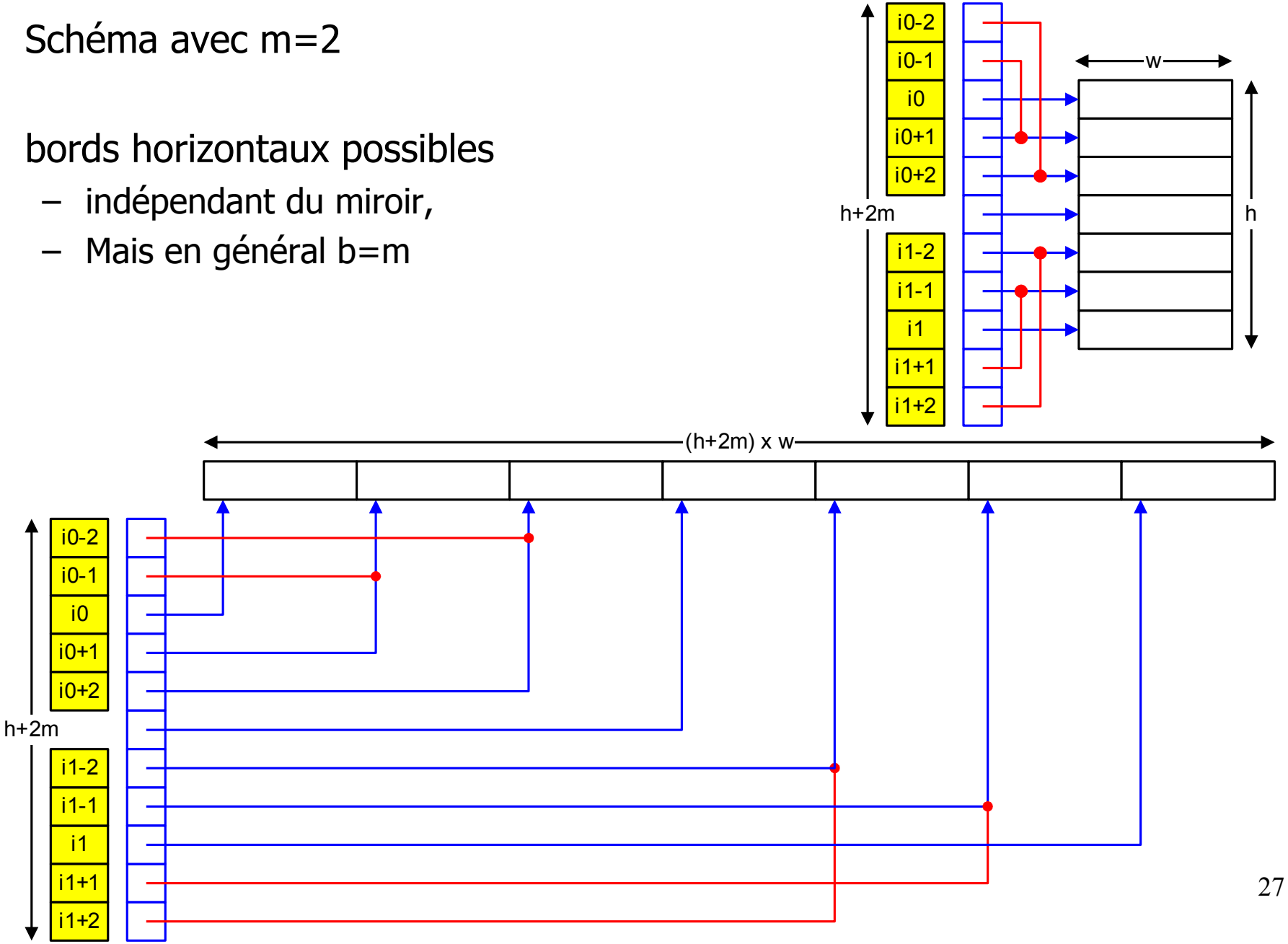
- bords horizontaux possibles
  - indépendant du miroir,
  - Mais en général b = c





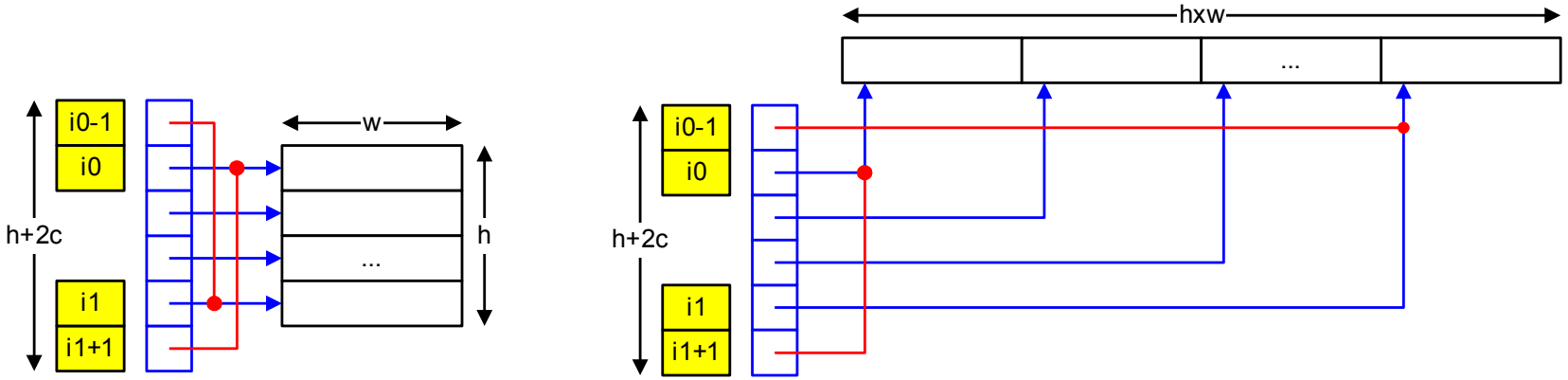
Matrice avec bords verticaux en miroir

- Schéma avec m=2
- bords horizontaux possibles
  - indépendant du miroir,
  - Mais en général b=m

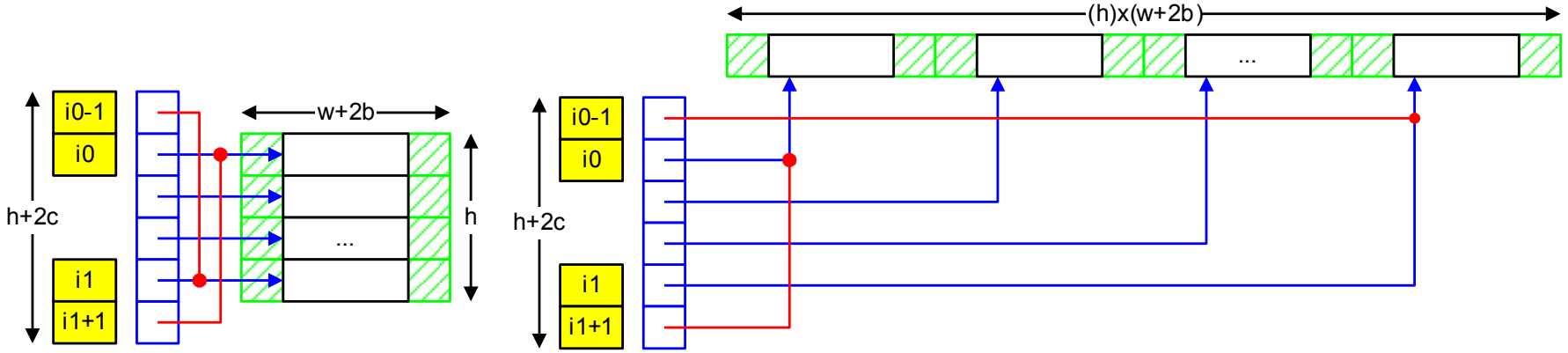


Matrices "cylindriques"

- Repliement cyclindrique de 1 sans bord (pour le jeu de la vie de Conway)

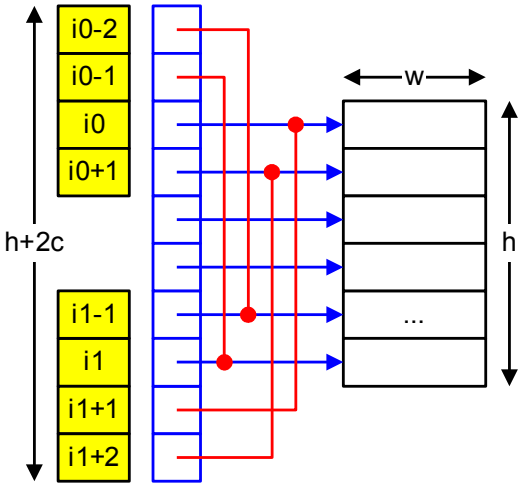


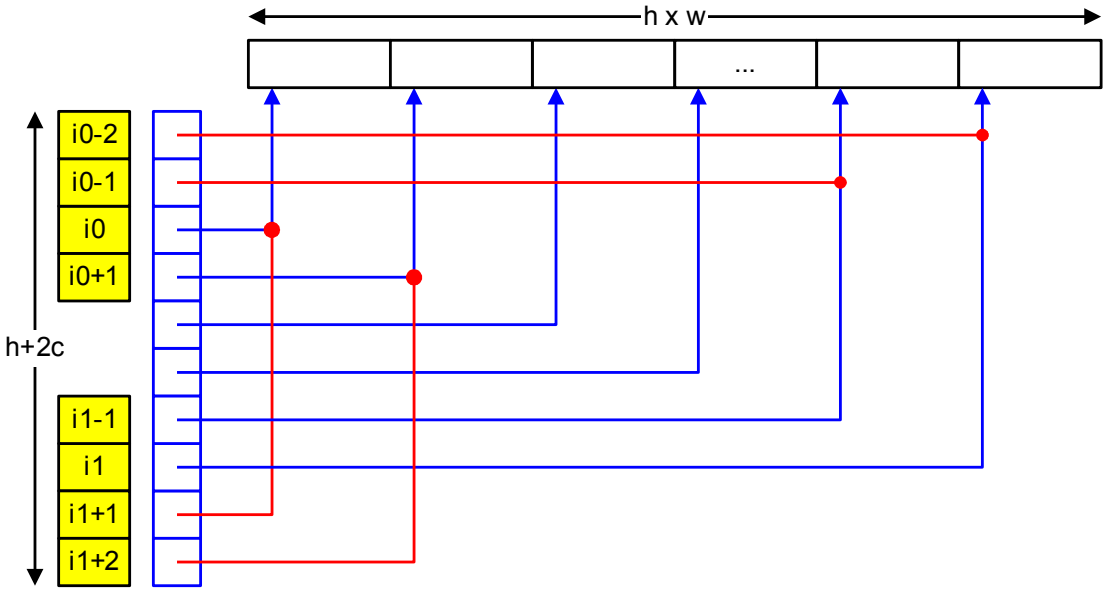
- Repliement cylindrique de 1 avec bords de 1



Matrices "cylindriques"

- Repliement cylindrique de 2





Conclusion: génie logiciel

- Cacher les fonctions d'allocation (et de désallocation) mémoire dans des fonctions d'interface pour contrôler et maîtriser l'allocation mémoire
  - Un bon code ne doit pas faire de malloc ou de free (new ou delete),
  - Mais doit faire appel à des fonctions le faisant et intégrant tout les tests nécessaire.
  - Cela permet d'ajouter un système de log, pour vérifier que toute la mémoire est bien désallouée à la fin du programme
- Il suffit de modifier 2 fonctions pour modifier l'allocation dans tout le programme
  - Par exemple ajouter du padding en fonction de l'OS (alignement 32 bits -> 128 bits)
- Le comportement des fonctions d'interface est modifiable et extensible
- Lisibilité améliorée, debug simplifié

Conclusion: architecture logicielle

- Possibilités quasi-illimitées: tout ce qui peut s'exprimer sous forme d'offset par rapport à une adresse de base
- Versions 3D non présentées en détails, mais bords, padding repliement, miroir, clamping et cylindre sont aussi possibles
- Extensible à 4D, … nD
  - L'expérience montre qu'il y a une perte de vitesse à partir de 4 à cause des multiples indirections => utiliser des pointeurs réduisant le nombre de dimensions
  - Les fonctions système manipulent des pointeurs (lorsqu'on ne peut pas faire autrement
  - Les fonctions de calcul (algorithmes) manipulent des tableaux, car cela améliore les capacités d'optimisation du compilateur (sémantique)