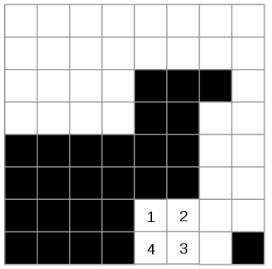
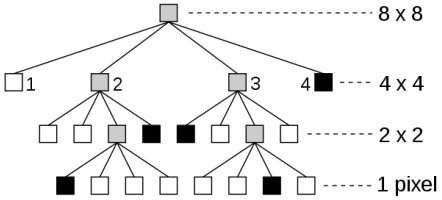
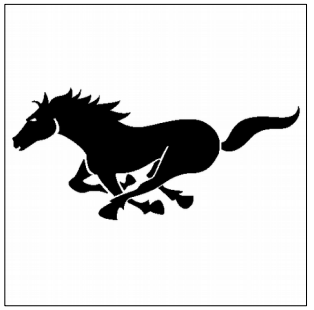
Exercice 2 : compression d'image





1) Récupérer la classe **QuadTree<T>** sur le site du cours (**quadtree.zip**), lire l'exemple associé. La considérer comme une bibliothèque : ne pas modifier le code.
2) Considérer une image noir&blanc carrée de taille une puissance de 2 (ex. d'image 512x512 fourni sur le site (**zipimage.zip**)) : la charger, l'afficher

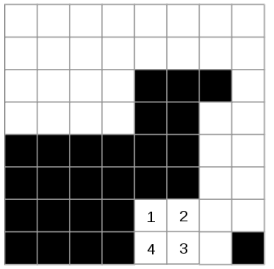


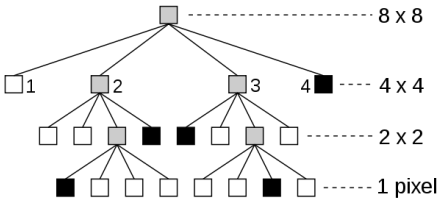
3) Encoder l'image dans un quadtree [≈ 30 LOC] :

algorithme :

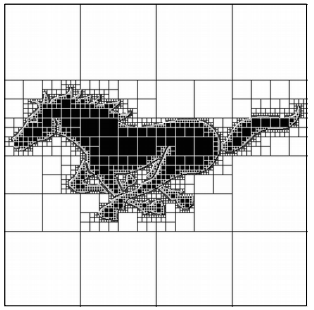
- descendre récursivement dans les sous-régions carrées de l'image
- **attention :** ne pas créer de sous-images, manipuler juste les coordonnées des régions
- **attention :** ne chercher à stoker les coordonnées de chaque région du quadtree, elles sont déduites du carré initial, calculées au vol quand on descend dans l'arbre
- construire le quadtree « <u>en remontant</u> » de la récursion :
  - quand on atteint une région de taille 1x1 (= un pixel)
    on retourne une feuille de la couleur du pixel
  - quand on reçoit 4 quadtrees correspondant à l'encodage des 4 sous-régions
    si ils ont la même couleur, c.-à-d. si ce sont 4 feuilles de même couleur
    on retourne une feuille de cette couleur
    sinon on retourne un nouveau nœud avec ces 4 quadtrees comme fils
- **attention :** ne pas tester explicitement l'uniformité de couleur de toute une région, chaque pixel ne doit être testé qu'un seule fois (pas $\log_2(512)$ fois)

Exercice 2 : compression d'image

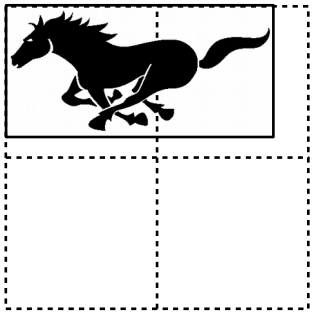




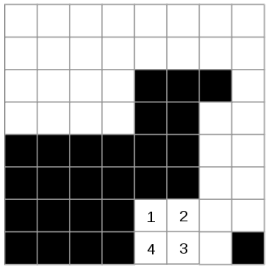
4) Optimiser : faire un *quaddag* qui partage toutes les feuilles Noir et toutes les feuilles Blanc
   - ne pas construire une nouvelle feuille blanche ou noire pour chaque pixel, n'en construire qu'une de chaque type avant l'encodage, dans des variables globales, et utiliser ces feuilles pré-construites lors de l'encodage d'un pixel
   - utiliser **protect_leaves_from_destruction** pour ne pas libérer ces feuilles partagées
5) Décoder le quadtree en tant qu'image [≈ 25 LOC] :
   - créer une nouvelle image de même taille, parcourir récursivement le quadtree en remplissant au vol les quadrants de la nouvelle image
   - **attention :** pour une décompression efficace, remplir un tableau et ne l'afficher en tant qu'image qu'à la toute fin
   - [option] dessiner des carrés dans l'image pour voir les grosses feuilles

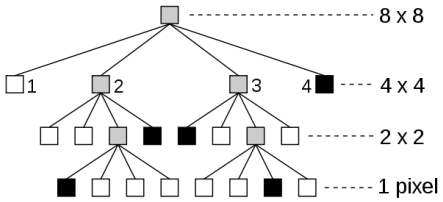


6) Compression :
   - estimer la taille de l'image compressée (liée au nb de nœuds du quadtree & **sizeof**)
   - comment mesurer légitimement le taux de compression ?
7) Traitement d'une image de dimension quelconque ≠ $2^N \times 2^N$ [≈ +10 LOC] :
   - arrondir les dimensions à la puissance de 2 supérieure
   - **attention :** compléter implicitement avec du blanc hors de l'image (ou du noir...), ne pas créer explicitement une nouvelle image carrée



Exercice 2 : compression d'image



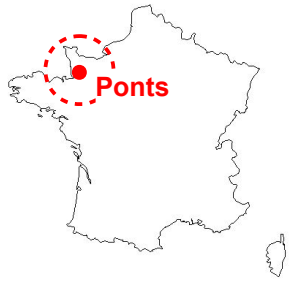


8) Traitement des niveaux de gris (compression avec perte) :
   - si 4 feuilles sont d'intensité voisine (différence d'intensité < seuil donné), les remplacer par une seule feuille d'intensité moyenne
   - faire varier le seuil et observer la dégradation de la qualité en fonction du taux de compression
   - variante pour éviter les dérives de dégradation : faire dépendre le seuil de la taille des régions (seuil ↘ quand région ↗)

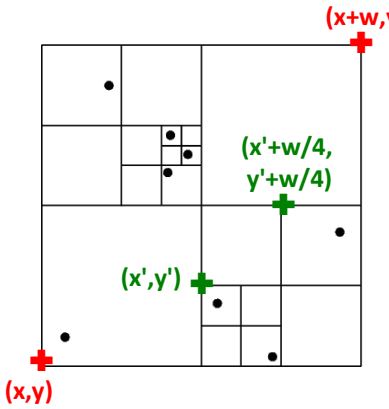
9) [Optionnel] Traitement de la couleur :
   - faire comme pour les niveaux de gris, indépendamment sur chaque canal RVB

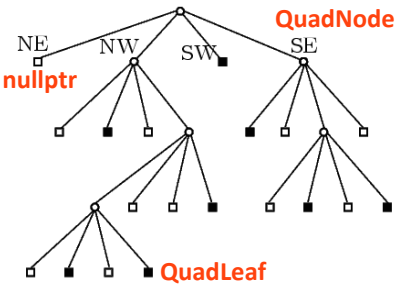
10) [Optionnel (long)] Taux de compression **effectif** :
   - implémenter une écriture du quadtree (image compressée) sur fichier
   - implémenter une lecture du quadtree à partir d'un fichier
   - calculer le taux de compression effectif :
     - rapport de taille entre le fichier avant compression et le fichier après compression
   - **attention :** la plupart des formats d'image sont déjà compressés, vous pouvez être déçus de vos résultats, sauf si vous considérez la compression avec perte...

Exercice 3 : quelle est la ville la plus proche de Ponts ?



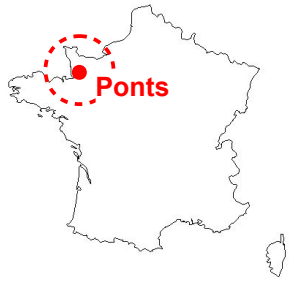
1. Récupérer les fichiers **quadtree.zip** et **villes1.zip** sur le site du cours. Lire les headers (.h).
   - **QuadTree<T>** : arbre à 4 branches avec un objet de type T aux feuilles
   - **Point2D<T>** : point dans $\mathbb{R}^2$ portant une information de type T
2. Coder le stockage de points 2D dans un quadtree, pour un intervalle carré donné de $\mathbb{R}^2$:
   - **indice 1 :** modéliser une feuille vide (sans point) par **nullptr**, modéliser une feuille avec un point 2D par un objet de type **QuadTree<Point2D<T>>***
   - **indice 2 :** les coordonnées des quadrants ne sont pas stockées dans l'arbre, elles sont déduites du carré initial, calculées au vol quand on descend dans l'arbre
   - **indice 3 :** considérer des carrés définis par un triplet (x,y,w), représentant les sommets (x,y)-(x+w,y+w). Voir pour ça au besoin le fichier **square.h**.





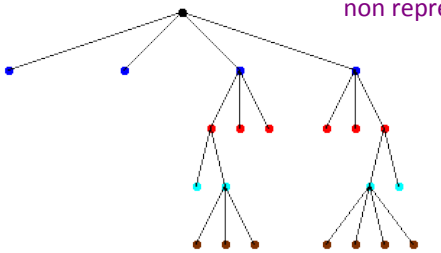
   - **indice 4 :** considérer les quadrants d'un carré, définis par une direction (NW,NE, SE, SW) et le sous-carré associé. Voir au besoin **quadrant.h**, implémenter **quadrant.cpp** [≈ 30 LOC]
   - **indice 5 :** voir l'interface de la fonction **insert** dans **neighbors.h** et la coder [≈ 40 LOC] cf. pp.64+
   - **attention :** si un point a les mêmes coordonnées qu'un point déjà dans l'arbre, il est à ignorer
   - **validation :** tester d'abord votre code avec qq points artificiels (affichage de l'arbre avec **display**)

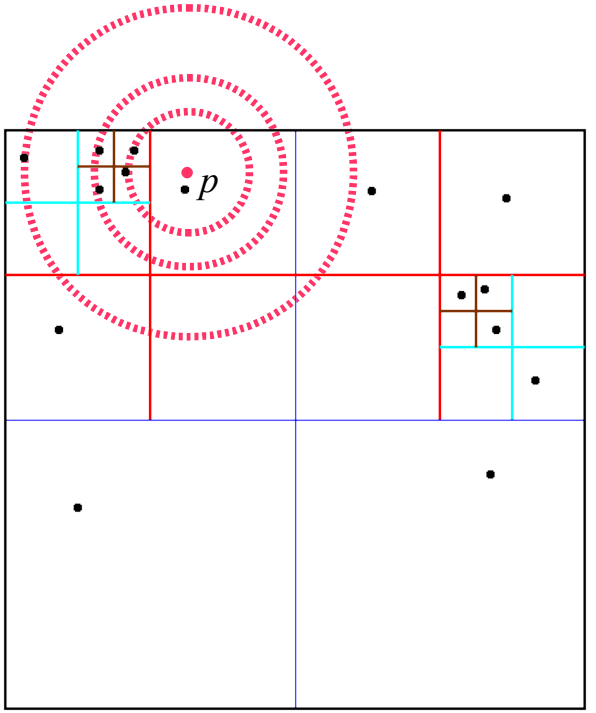
Exercice 3 : quelle est la ville la plus proche de Ponts ?



3. Implémenter, pour la distance euclidienne, la recherche de tous les proches voisins d'un point p donné, c.-à-d. à une distance $\leq r$ donné :
   - **algorithme :** descendre récursivement dans l'arbre (cf. p. 69)
   - **indice 1 :** implémenter une fonction **intersects_disk** qui teste si un carré donné intersecte un disque centré sur p de rayon r (cf. **square.h**) [≈ 20 LOC]
   - **astuce :** éviter sqrt, comparer des distances^2
   - **indice 2 :** voir l'interface de la première fonction **search** dans **neighbors.h** et la coder [≈ 40 LOC]
   - **validation :** vérifier sur quelques points
4. Ajouter la recherche du plus proche voisin :
   - **algorithme :** modification au vol de r (cf. p. 71+)
   - **attention :** ne pas dupliquer de code, ajouter un bool à votre fonction de recherche des proches voisins
   - **indice :** voir l'interface de la deuxième fonction **search** dans **neighbors.h** et la coder [≈ +5 LOC]
   - **validation :** vérifier sur quelques points

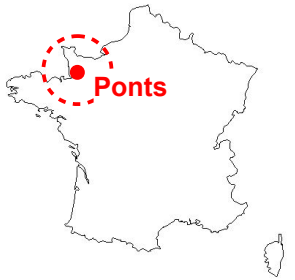
(branches à feuilles nulles non représentées)





© James Demmel

Exercice 3 : quelle est la ville la plus proche de Ponts ?

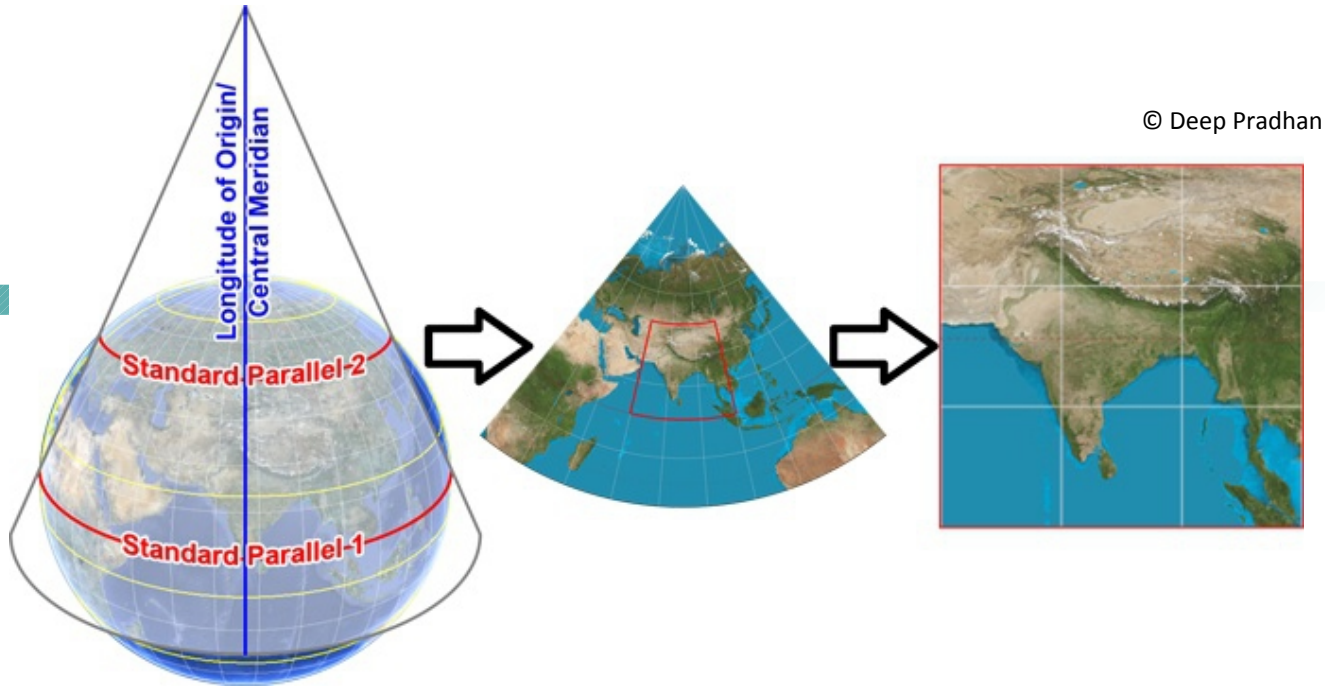


5. Charger les villes de France métropolitaine (**villes.txt**) dans **QuadTree<Point2D<Town>>* :**
   - lecture du fichier, dimensionnement du carré initial : voir **read_file** dans **town.h**, **town.cpp**
   - **attention :** les distances en latitudes-longitudes n'ont pas de sens, passer en Lambert93 (p. 89)
6. Quelle est la taille du quadtree ?
   - **indice :** cf. fonctions **nLeaves**, **nNodes**, **nTrees** de **QuadTree**
7. Quelle est la ville la plus proche de **Ponts** ?
   Combien de nœuds du quadtree faut-il parcourir pour trouver ça ?
   Comparer avec un parcours linéaire du vecteur de villes.

   ☺ Chercher l'adresse de la mairie de Ponts...
8. Quel est le temps moyen pour trouver une plus proche ville :
   (a) avec un quadtree ?
   (b) avec un vector ?
   - **indices :** tirer 100 villes au hasard dans le vecteur de villes (mesure du temps: voir **example.cpp**)
9. Combien de recherches de plus proches villes faut-il faire en moyenne pour rentabiliser le temps de construction du quadtree ?
10. [Bonus optionnel] Prendre en compte le fait que certaines villes, du fait des arrondis, ont les mêmes coordonnées : les conserver toutes au lieu de ne garder que la première (cf. 2)

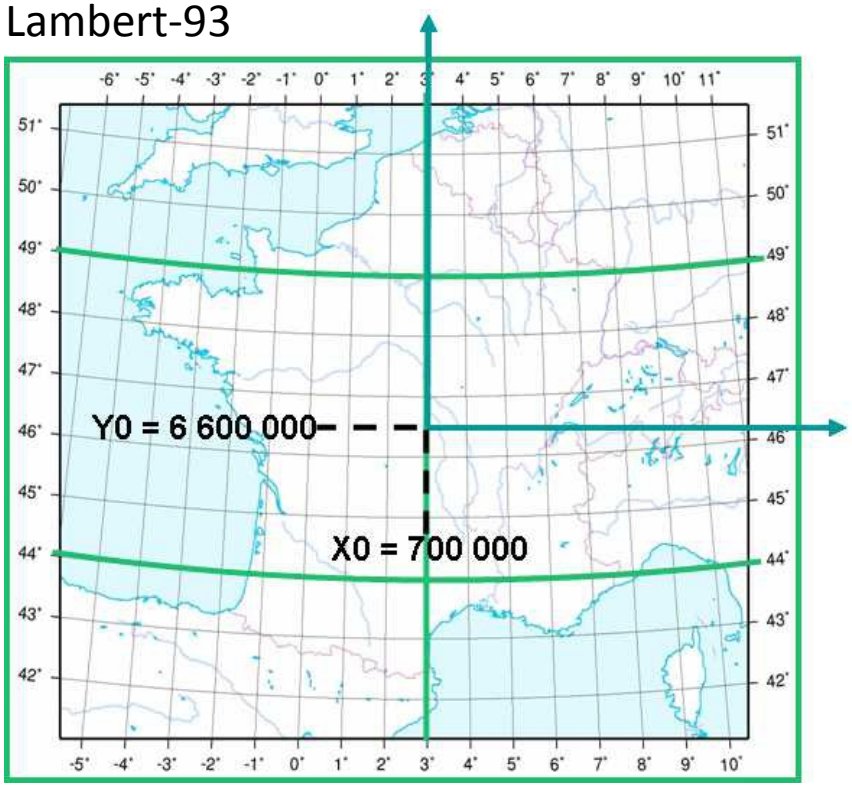
La carte et le territoire…

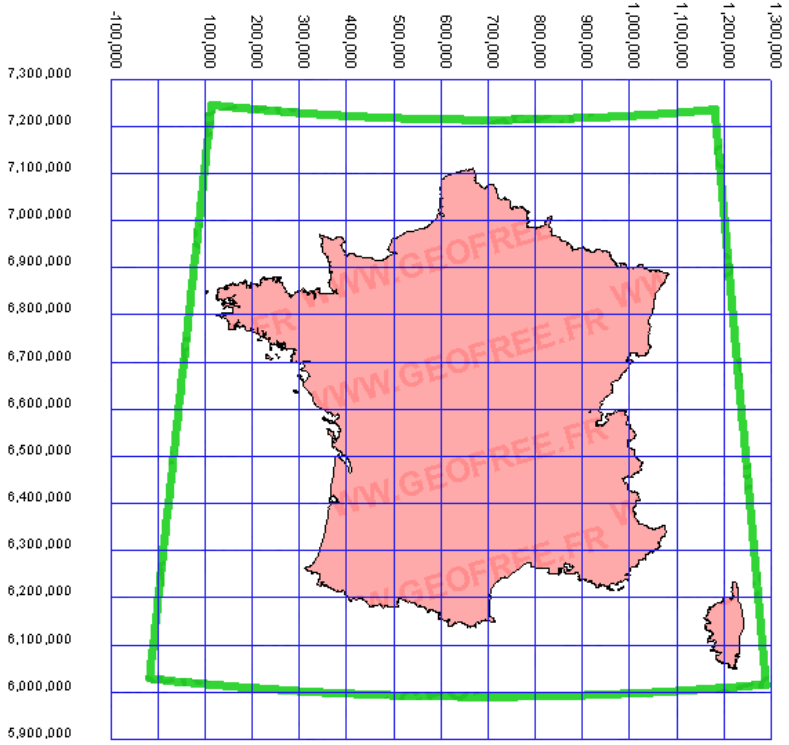
Projection conique conforme de Lambert



© Deep Pradhan

Lambert-93





© IGN

© geofree.fr

Bibliothèques

- Bibliothèque STL (structures de données courantes)
  - structures de données : vector, list, stack, queue, set...
  - http://www.cplusplus.com/reference/stl/
- Bibliothèque Imagine++ (images, graphisme)
  - site : http://imagine.enpc.fr/~monasse/Imagine++/
  - quick start :
    - http://imagine.enpc.fr/~monasse/Stereo/quickStartImagine++.pdf
    - http://imagine.enpc.fr/~monasse/Info/programmer.pdf (annexe B)
- Base pour construire des quadtrees
    - http://imagine.enpc.fr/~marletr/enseignement/proal/quadtree.zip