Exercice 1

La fonction présentée au cours s'arrête lorsque $|x - y^2| < c$, où c est une constante (p.ex. 0.001).

- Avec cette condition d'arrêt, l'incertitude tolérée est une constante (c). Elle est donc indépendante de l'ordre de grandeur du nombre dont on recherche la racine. Pour les très petits nombres, l'incertitude devient alors beaucoup plus grande que le nombre lui-même. Sur un exemple, on comprend facilement qu'obtenir la racine du nombre 2 avec une incertitude de 10 n'a pas grand sens.
- Un nombre flottant a un nombre limité de chiffres significatifs. Par conséquent, soit deux nombres flottants sont égaux, soit leur différence est minorée par un nombre d'autant plus grand qu'ils le sont aussi. Dans notre cas, si x et donc y^2 sont très grands, le seul moyen de satisfaire la condition est que $x = y^2$, ce qui peut ne jamais arriver.

Un bon moyen pour contourner ces problèmes est de calculer la variation relative de y : $|y - y'|/y'$, où y et y' sont deux approximations successives. Cette méthode a l'avantage de fonctionner quelle que soit la fonction à approximer, on n'aura donc pas besoin de la changer pour l'exercice 2.

Nouvelle version :

```
def sqrt(x: double): double = {
  def sqrtIter(prev: double, guess: double): double =
    if (isGoodEnough(prev, guess)) guess
    else sqrtIter(guess, improve(guess));

  def improve(guess: double) = (guess + x / guess) / 2;

  def isGoodEnough(prev: double, guess: double) =
    abs(prev - guess) / guess < 0.001;

  sqrtIter(1.0, improve(1.0));
}
```

Attention à ne pas donner la même valeur pour `guess` et `prev` (lors du premier appel) sans quoi `isGoodEnough` renverrait immédiatement `True` sans qu'aucune itération n'ait eu lieu.

Il est aussi possible de ne passer qu'un seul paramètre à `sqrtIter` en plaçant l'approximation suivante dans une valeur locale :

```
  def sqrtIter(guess: double): double = {
    val next = improve(guess);
    ...
  }
```

Exercice 2

On peut garder le même code qu'avant, il suffit de changer `improve` :

```
def improve(y: double) = (x / (y * y) + 2 * y) / 3;
```

Attention, si vous utilisez la fonction `isGoodEnough` originale, qui compare le carré de y avec x, il faut aussi la changer (pour qu'elle compare le cube de y avec x, bien sûr).

Scala ne possède pas d'opérateur d'élévation à la puissance. Pour élever une valeur au cube il faut donc écrire $y * y * y$ ou passer par Java : `java.lang.Math.pow(y,3.0)`.

Exercice 3

Comme dit à la séance d'exercices, la fonction doit prendre en argument le numéro de la colonne et le numéro de la ligne, le triangle étant décalé afin que les 1 de gauche forment une ligne verticale.

	0	**1**	**2**	**3**	**4**
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Pascal est alors défini comme suit (c : colonne, l : ligne) :

$$\text{pascal}(c,\ l) = \begin{cases} 1 & \text{si } c = 0 \\ 1 & \text{si } c = l \\ \text{pascal}(c-1,\ l-1) + \text{pascal}(c,\ l-1) & \text{sinon} \end{cases}$$

L'exercice ne précise pas ce qu'il faut renvoyer si c n'est pas dans l'ensemble $[1; l]$, on peut donc supposer que ce cas ne se produit pas. L'implémentation en Scala ci-dessous renvoie 1 dans ce cas.

```
def pascal(c: int, l: int): int =
  if (c <= 0 || c >= l) 1
  else pascal(c - 1, l - 1) + pascal(c, l - 1);
```

Prenez garde au fait que lorsqu'on demande une fonction *récursive*, cela signifie qu'on ne veut pas une solution impérative basée sur les boucles. D'ailleurs, dans Scala comme dans beaucoup d'autres langages fonctionnels, *le concept de boucle n'existe pas*. Il est en effet toujours possible de remplacer des boucles par des fonctions récursives, et même de définir directement dans le langage des fonctions simulant des constructions de boucles classiques, comme `while`.

Notez finalement qu'on ne demandait pas de construire un tableau.