INFORMATIQUE

REVISIONS

Fonctions

TD : Recherche du zéro d'une fonction

- **Méthode de Newton**
- **Méthode par Dichotomie**

Fonction et modularité

L'intérêt premier des fonctions est d'éviter la **duplication de code** :

à chaque fois que l'on fait la même suite d'opérations on remplacera avantageusement cette portion de code par un appel à une fonction qui peut dépendre d'arguments ou paramètres variables :

Moins de code / Plus de lisibilité

Il est aussi plus facile de copier /coller une fonction qui servira dans un autre projet
voire de l'importer depuis un module : c'est le paradigme de la **modularité** en informatique

En programmation, l'intérêt des fonctions va bien au delà de la non duplication de code :

- Certains algorithmes sont difficiles à implémenter sans fonction [récursivité]

- Un projet complexe nécessite une forte modularité pour être efficace et compréhensible :
  L'enjeu n'est plus tant le codage que la structuration en fonctions élémentaires du projet.
  Le travail collectif est impensable sans modularité.

- En programmation objet cette logique se poursuit avec les classes et les méthodes (fonction). Sur chaque objet agissent un certain nombre de fonctions [maListe.sort()]

Logique Client - Serveur

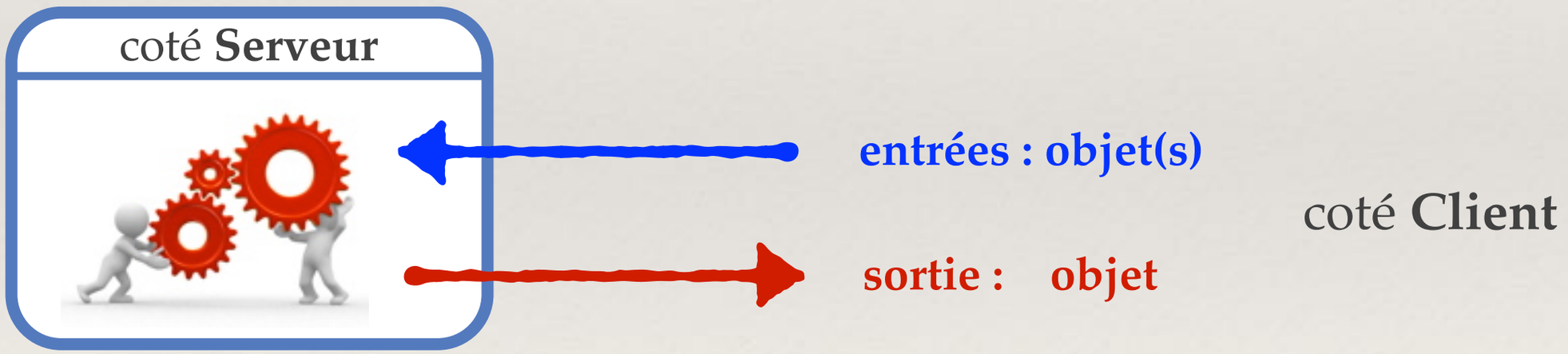
On désigne par **Client** le fait de faire appel à une fonction : qui tient le rôle de **serveur**

<u>En tant que client :</u>

- **je n'ai pas besoin et <u>je ne veux pas savoir</u> comment procède la fonction**
- je me repose sur le fait que la fonction fera ce pour quoi elle est conçue.
- je dois seulement donner en argument les objets attendus par la fonction.

<u>En tant que serveur :</u>

- je veut optimiser mon algorithme pour répondre au client de la façon la plus efficace.
- **je veux pouvoir changer l'algorithme interne sans que cela n'affecte l'extérieur**
- je peux devenir client d'une autre fonction pour produire le résultat



En pratique on est souvent à la fois Client et Serveur tour de rôle, au sein de son propre projet, et très souvent Client de module à disposition.

Plus le projet est long plus il est avantageux de le modulariser, il faut alors bien avoir conscience du rôle que l'on joue vis-à-vis d'une fonction.

Logique Client - Serveur

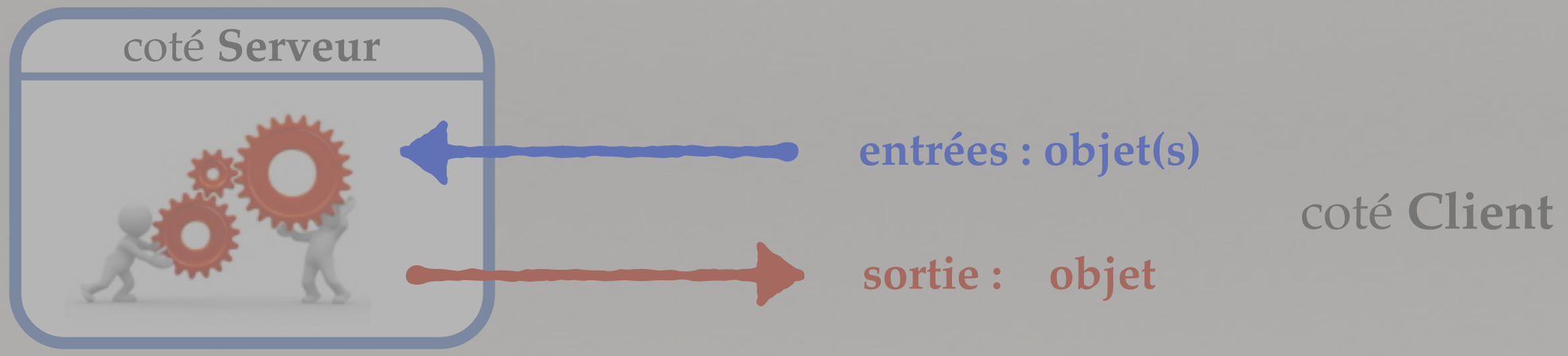
On désigne par **Client** le fait de faire appel à une fonction : qui tient le rôle de **serveur**

En tant que client :

- Je peux **penser le PB dans sa globalité** sans me soucier des détails techniques -> bien plus facile !

En tant que serveur :

- Je dois juste résoudre **un point technique** sans me soucier du reste -> beaucoup plus facile !



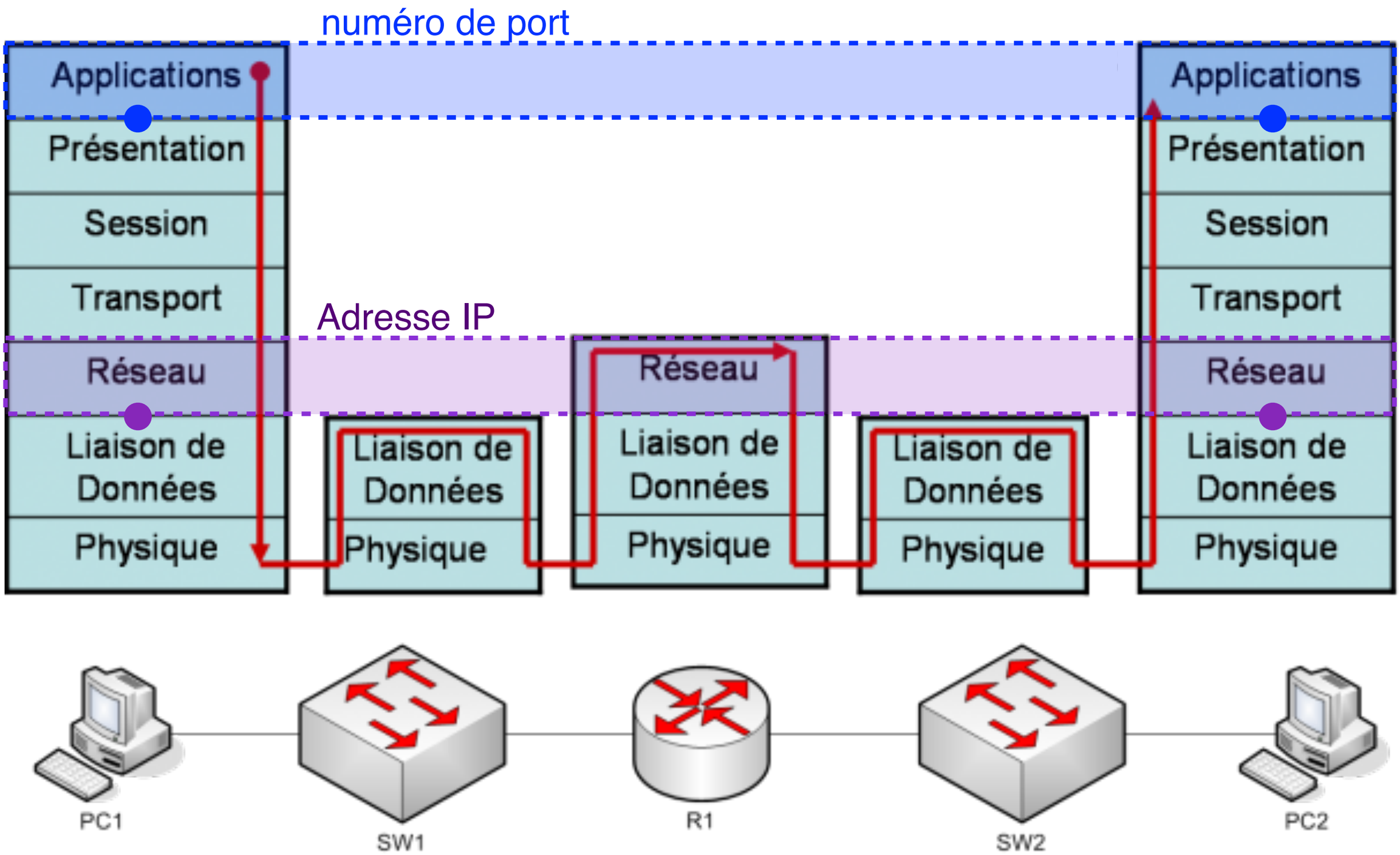
En pratique on est souvent à la fois Client et Serveur tour de rôle, au sein de son propre projet, et très souvent Client de module à disposition.

Plus le projet est long plus il est avantageux de le modulariser, il faut alors bien avoir conscience du rôle que l'on joue vis-à-vis d'une fonction.

La logique Client / Serveur est très générale en informatique consiste à déléguer tout ce qui ne relève pas de ma compétence en propre : **permet une forte adaptabilité des systèmes.**

=> je donne les informations nécessaires (passe commande : entrées)

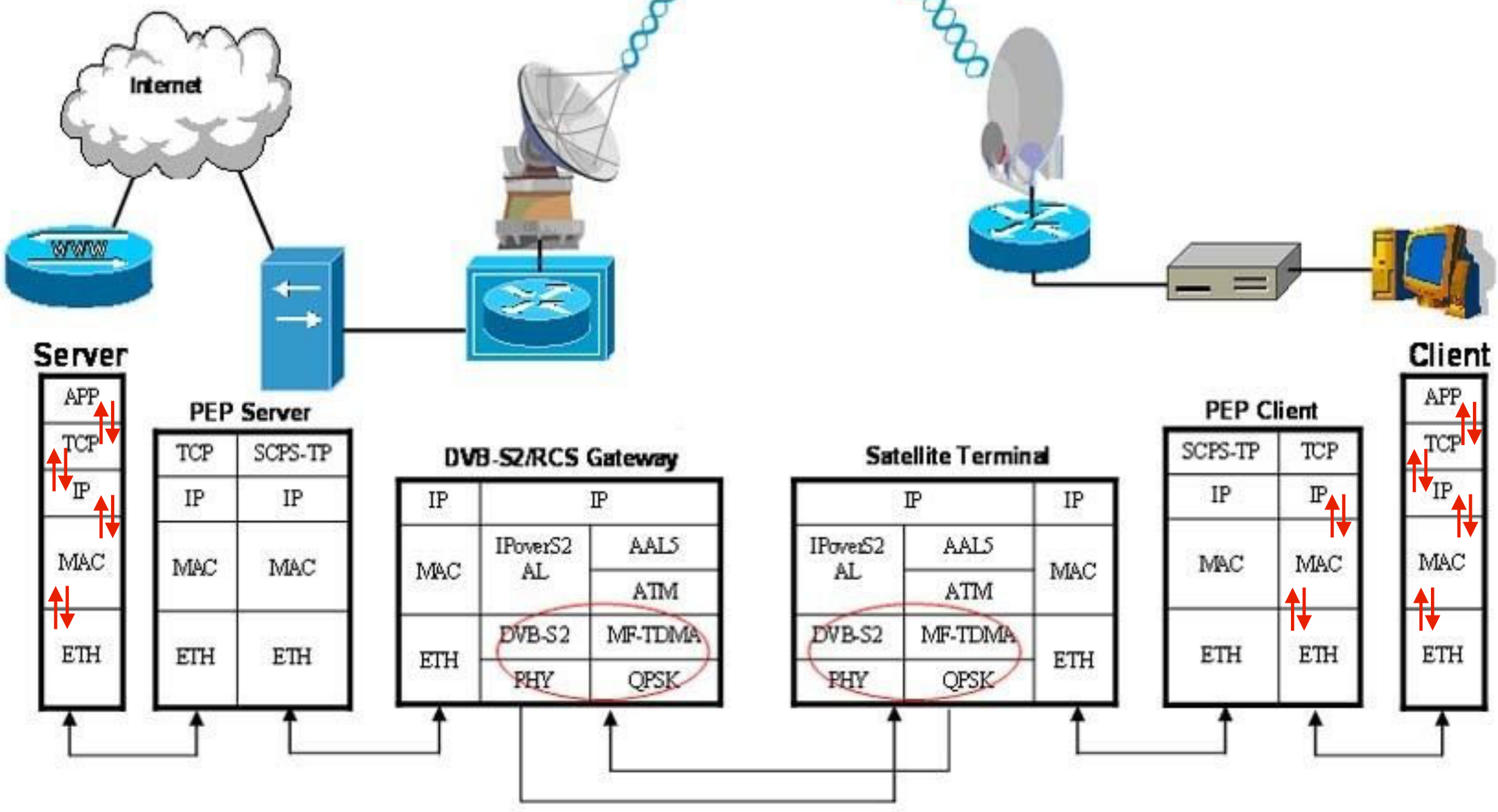
=> j'attends en retour le résultat du serveur (réception de la commande : sortie)



Pour franchir chaque étage : Logique Client /serveur

sortie

entrées



Conception d'une fonction

On pensera toujours la conception d'une fonction dans le cadre Client / Serveur

Structure globale du projet / Travail collectif

1 - Quels arguments doit-on passer à la fonction [c-à-d quels objets] ?

2 - Quel objet veut on récupérer en sortie de fonction ? [quelle structure de données]

Partie Technique / Algorithmique

3 - Rédiger le corps de fonction **TOUJOURS A LA FIN**
pour reconnecter les entrées et les sorties

Notion de callable

Un «callable» est littéralement un objet que l'on peut appeler : autrement dit un serveur !
En python ce sera donc une **fonction** ou une **méthode de classe**, et on l'appelle en passant des arguments **entre parenthèses** **>>> maFonction(x)** **>>> maListe.sort()**

- **L'objet fonction, sans parenthèse donc sans appel, est de type fonction**
  >>> type(maFonction) donnera « function » c'est un callable.
  >>> type(maListe.sort) donnera <class 'builtin_function_or_method'>

- En revanche **l'appel à la fonction, donc avec parenthèse** et argument si nécessaire, **est du type de l'objet que retourne la fonction**.
  >>> type(maFonction(x)) dépend de la fonction

- **Pile d'appels fonctionnels :**
  En réalité type(maFonction(x)) n'évalue pas maFonction mais le résultat produit par maFonction qui agit sur x.

  A nouveau on lit les appels fonctionnels de droite à gauche :
  - **x** est évalué puis passé à **maFonction**
  - **maFonction**(**x**) est évalué puis passé à la fonction **type**
  - **type(maFonction(x))** est enfin évalué.

- **Pile d'appels fonctionnels :**

En réalité type(maFonction(x)) n'évalue pas maFonction mais le résultat produit par maFonction qui agit sur x.

Ex :

```
>>> def plus(a, b):
...     return a+b
...
```

```
>>> type(plus(1, 2))
<class 'int'>
```

```
>>> type(plus(1, 2.))
<class 'float'>
```

```
>>> type(plus('B', 'A'))
<class 'str'>
```

Conséquences du typage dynamique :

```
>>> type(plus((1,2), (3,)))
<class 'tuple'>
```

```
>>> type(plus(True, False))
<class 'int'>
```

Rq :

```
>>> type(plus)
<class 'function'>
```

plus est un objet fonction (qui sont des callables)

Les « scopes » et la règle LEG

La **portée des variables** en python, c'est à dire l'étendue du code au sein de laquelle un nom de variable peut-être reconnu est régie par la règle dite LEG.

Local

Environment

Global

- On chercher d'abord **localement** la variable dans le corps de fonction.
- Si on ne l'y trouve pas, dans l'**environnement** fonctionnel [pile d'appel fonctionnel]
- Si on ne l'y trouve toujours pas, dans le module **global** du programme.

(Rq :Voir le complément associé : c'est un sujet en soit)

Ne pas confondre la variable x du module Global avec celle x du scope de la fonction

```
1  x=3
2
3  def f(x):
4      x=4
5      return x
6
7  print(x)
8  print(f(x))
9  print(x)
```

Affichage :
3
4
3

Deux erreurs classiques qu'il ne faut plus faire :

Erreur 1 —> Oublier d'appeler la fonction

En soit la fonction n'est qu'une définition, et le corps de texte de la fonction n'est que potentiellement actif : si on déclenche l'appel à la fonction.

En revanche l'objet fonction est un « callable » :

=> Si vous ne l'appelez pas, en lui passant les arguments nécessaires,
Rien ne se passe et c'est normal !

=> Si il y a une erreur, celle-ci ne se produira que lorsque le code sera effectivement interpréter c-à-d lors de l'appel : « **runtime error** ».

Enfin, bien réaliser que toutes les variables créées dans le bloc fonctionnel seront détruites lorsque la fonction retourne sont résultat : pouff !

Rq : pour une bonne gestion de la mémoire, leur triste destin est d'être avalées par le ramasse miette !

Erreur 2 —> Confondre return et print

Erreurs récurrentes :

- **Penser que « return » va afficher quelque chose**
- **Penser que print() renvoie le résultat.**

Conseil : **Efforcez-vous d'oublier l'affichage dans la console. Ce n'est qu'un outil. Le programme n'est pas là …. les objets encore moins.**

return clôt le bloc fonctionnel en déterminant l'objet que renvoie la fonction.

- En l'absence de return la fonction renvoie l'objet «**None**».
- **Tout ce qui suit la commande return n'existe pas.**

Print() est une fonction en elle même, qui provoque l'affichage sur la console. Elle a un rôle analogue à **plt.show()** méthode qui déclenche l'ouverture d'une fenêtre graphique.

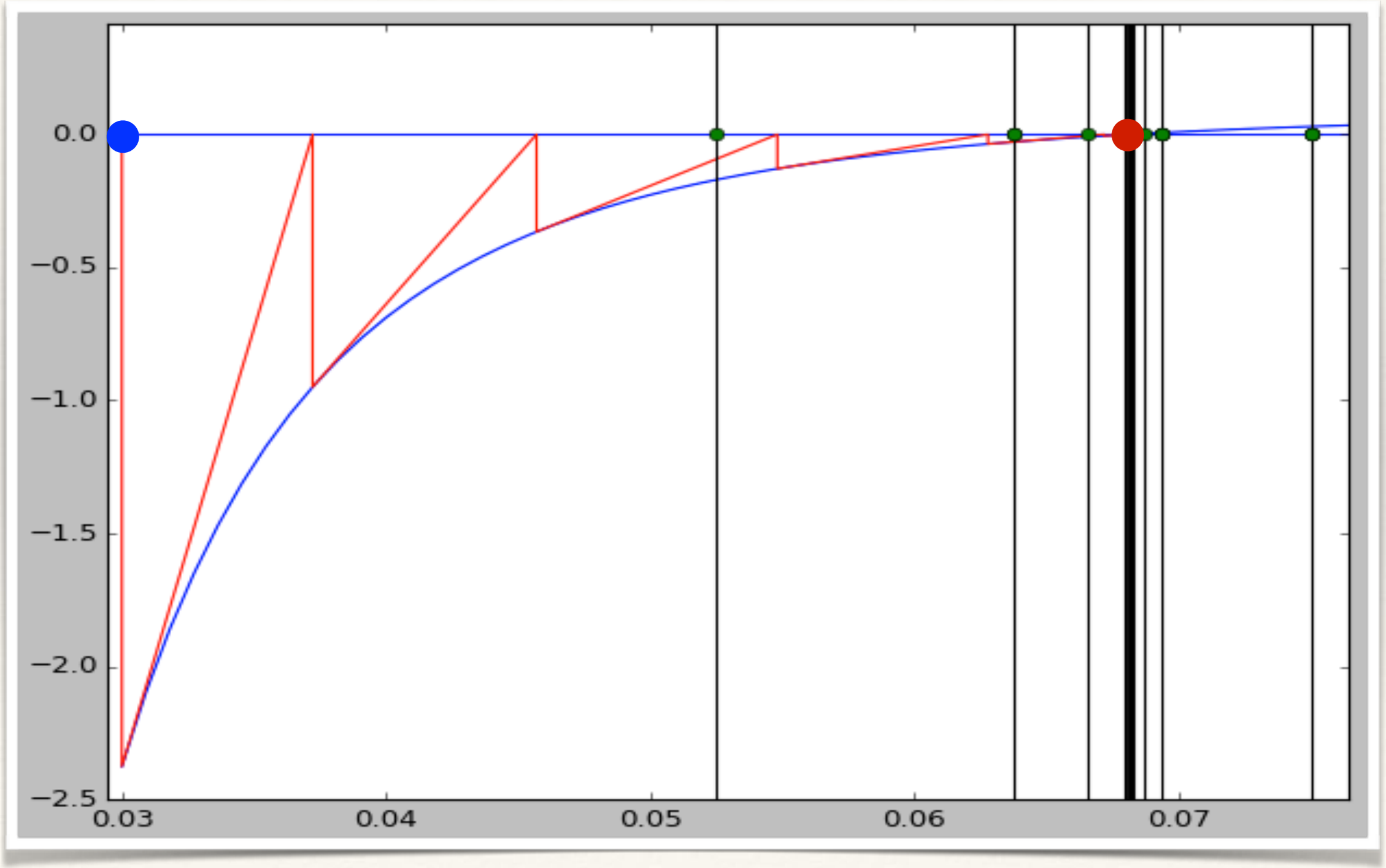
La fonction print ne renvoie rien au sens des fonctions [elle renvoie **None**].

```
>>> a=print("Toto")
Toto
>>> a

>>> type(a)
<class 'NoneType'>
```

TD : Recherche du zéro d'une fonction

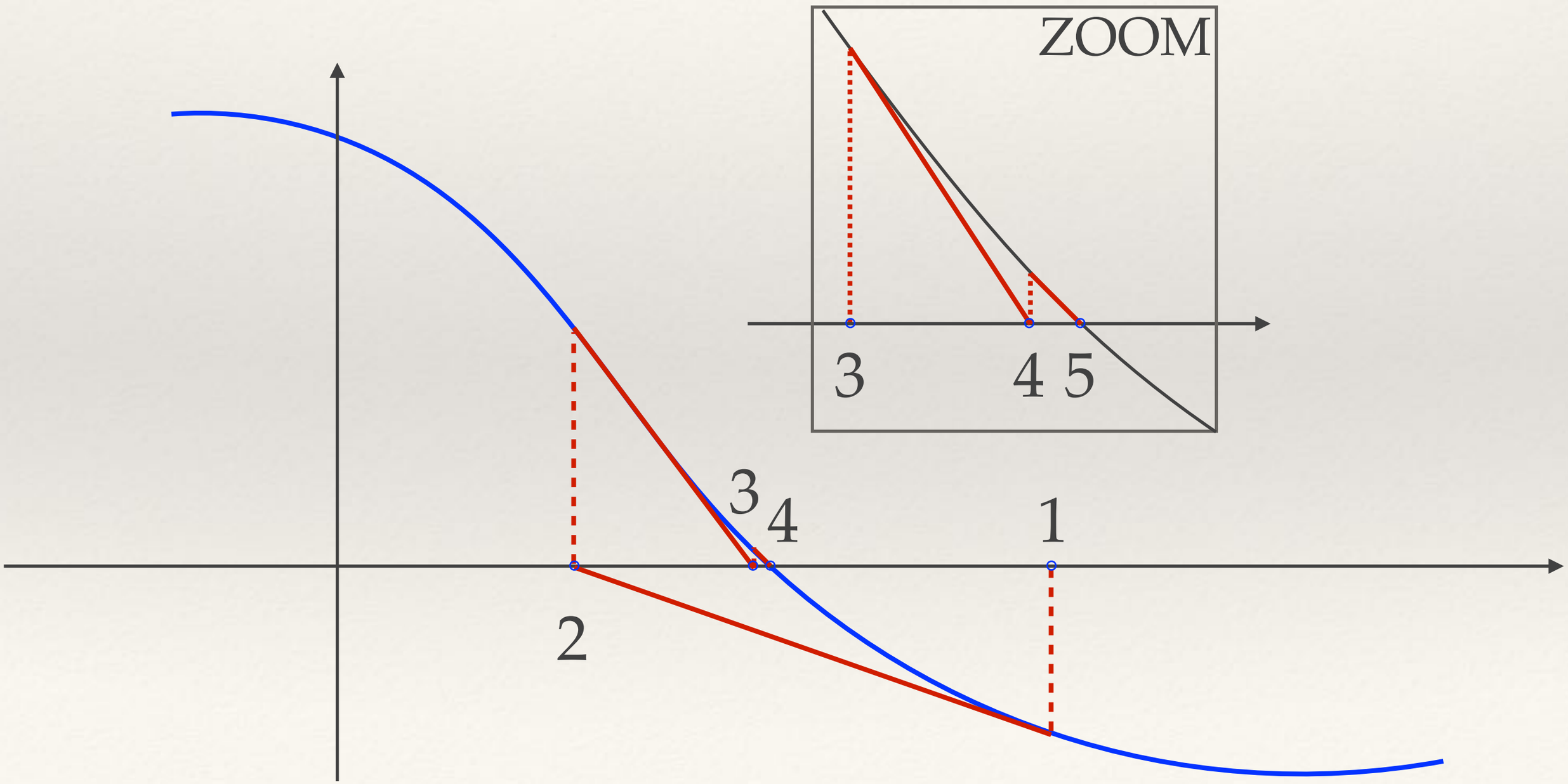
Méthode de Newton - Méthode par dichotomie



TD Recherches de Zéros

Méthode de Newton et méthode par dichotomie

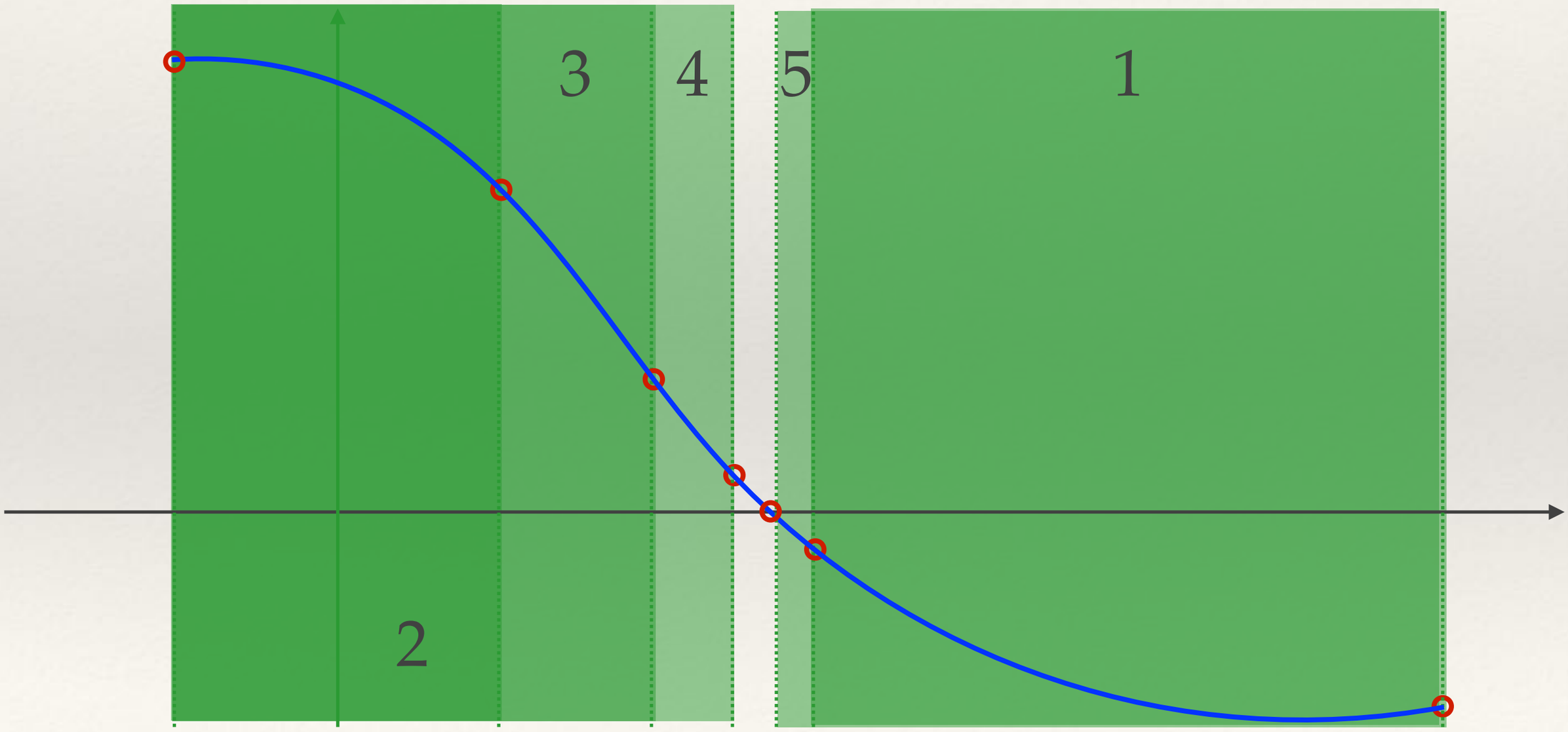
La **méthode de Newton** consiste partant d'une abscisse arbitraire, à extrapoler la tangente en ce point pour trouver l'intersection avec l'axe des abscisses en espérant se rapprocher du zéro rapidement. On ré-itère cette opération jusqu'à atteindre une convergence suffisante !



TD Recherches de Zéros

Méthode de Newton et méthode par dichotomie

La **méthode par dichotomie** consiste à prendre deux points de départ xMin et xMax de signes opposés : on coupe alors l'intervalle en deux et on regarde le signe. On réduit ainsi la taille de l'intervalle de moitié en ne gardant que la moitié où le signe change.
On ré-itère cette opération jusqu'à atteindre une convergence suffisante !



Diviser pour Régner !!!

Conclusion sur la méthode de Newton

La méthode de Newton constitue une excellente méthode de principe, et converge beaucoup plus rapidement vers un zéro pour des problèmes bien circonscrits :
- Equation dérivable et sa dérivée connue.
- Domaine de définition de la fonction et de sa dérivée bien établi.
- Lorsqu'on sait que la convergence est assurée.
- Unicité du zéro.

ex : Recherche de la racine carrée [$X_0 > 0$ suffit à garantir une convergence rapide]
=> C'est la méthode utilisée sur une calculatrice. Elle se généralise aux racines n-ièmes.

La grande force de la méthode de Newton est de tirer parti de la connaissance de la dérivée : c'est aussi son talon d'Achille !!! car on ne peut pas faire sans.

<u>En effet cette méthode perd tout son intérêt lors de l'étude d'un signal physique :</u>

- Pour une version numérique, il faut une dérivée calculable donc un signal très régulier [absence de bruit]. Il faut de plus interpoler sur le signal échantillonné.
- On risque vite de sortir du domaine du signal.
- Il n'y a pas de garantie de convergence.
- On ne sait pas à l'avance comment choisir les conditions de départ.

Conclusion sur la méthode par Dichotomie

La méthode de recherche de zéro par dichotomie est certes moins rapide que la méthode de Newton mais elle reste très rapide et s'applique à une très grande variété de problèmes informatiques.

- La méthode ne nécessite pas de connaitre une expression analytique.
- Pas besoin de la dérivée non plus.
- L'algorithme ne peut pas sortir du domaine de définition

On peut la mettre en place sur un signal physique même bruité.

Remarques :

- Pour un signal bruité, la question de la multiplicité des zéros se pose
- On obtient pas une valeur, mais un encadrement du zéro.

- En informatique :
  La recherche par Dichotomie est intéressante pour des fonctions strictement monotones. On peut alors trouver tout élément dans un temps logarithmique :
  C'est le meilleur algorithme de recherche dans un tableau trié.

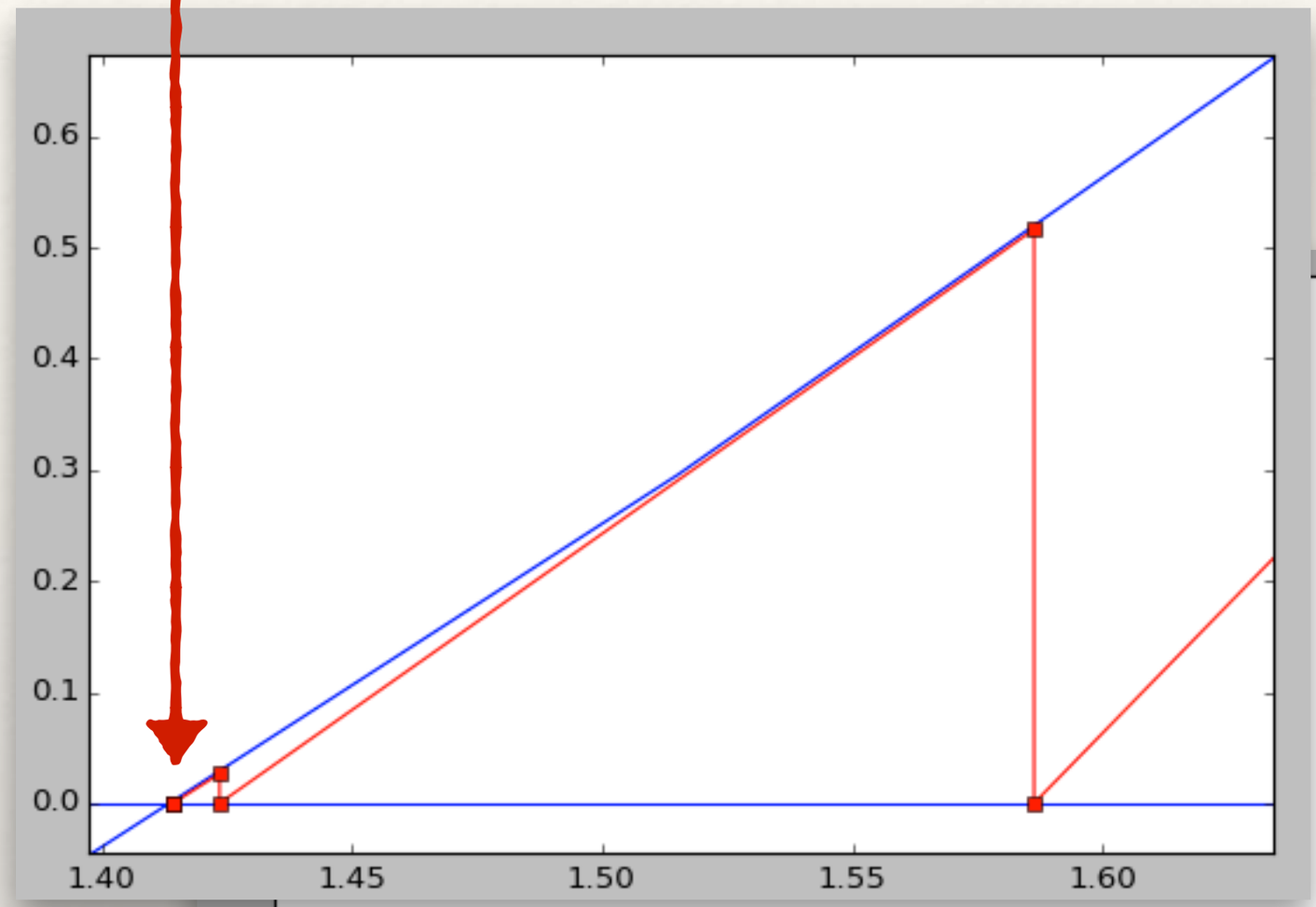
Algorithme de Babylone

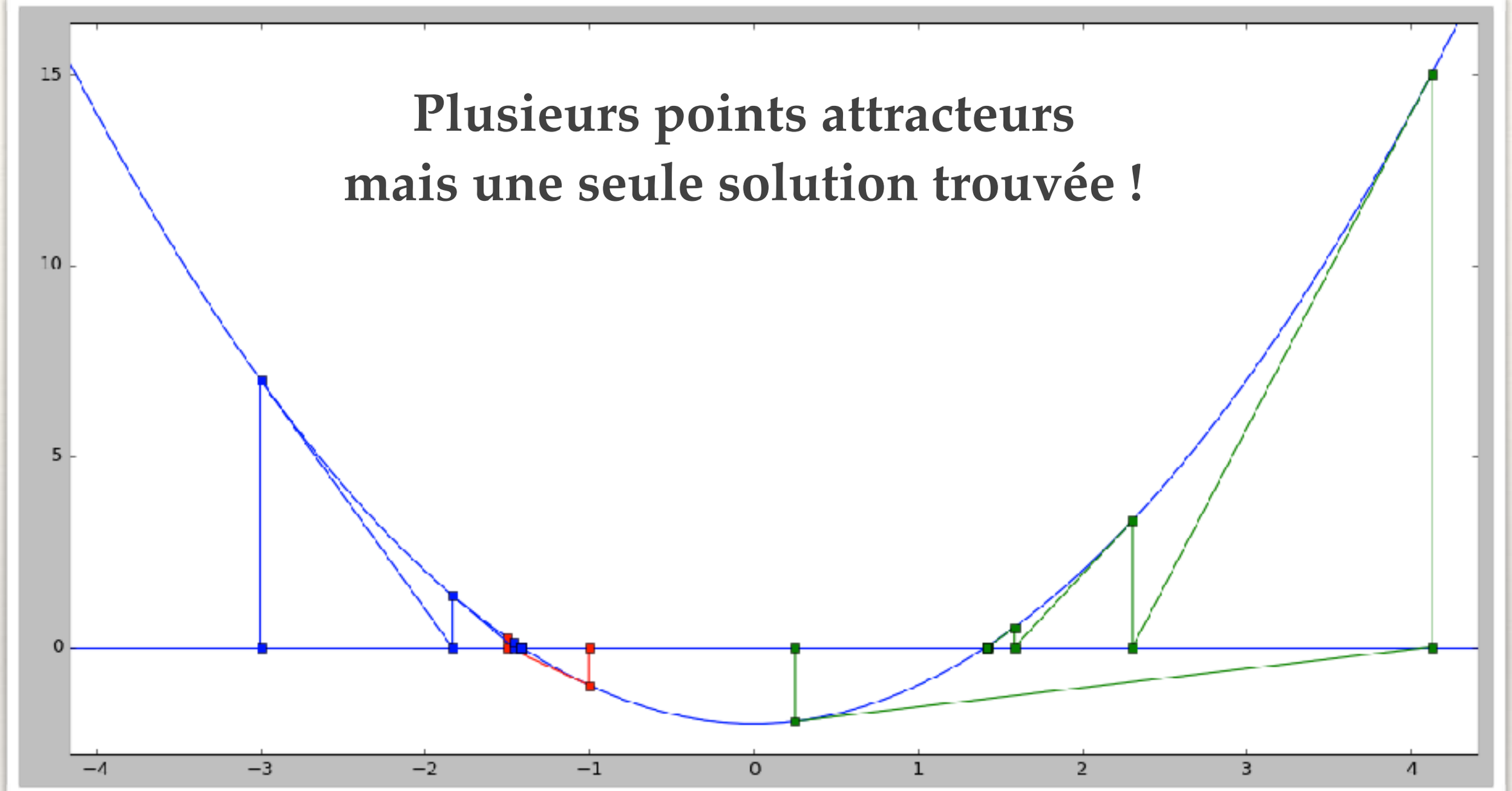
Application à la recherche de la racine carrée

PB antique : relatif à la répartition des terres arables

Etant donner que chacun s'est vu attribuer la même surface à cultiver,
comment trouver la longueur a donner à la parcelle pour qu'elle soit carrée ?

$$\sqrt{2} = 1.414213562373095$$

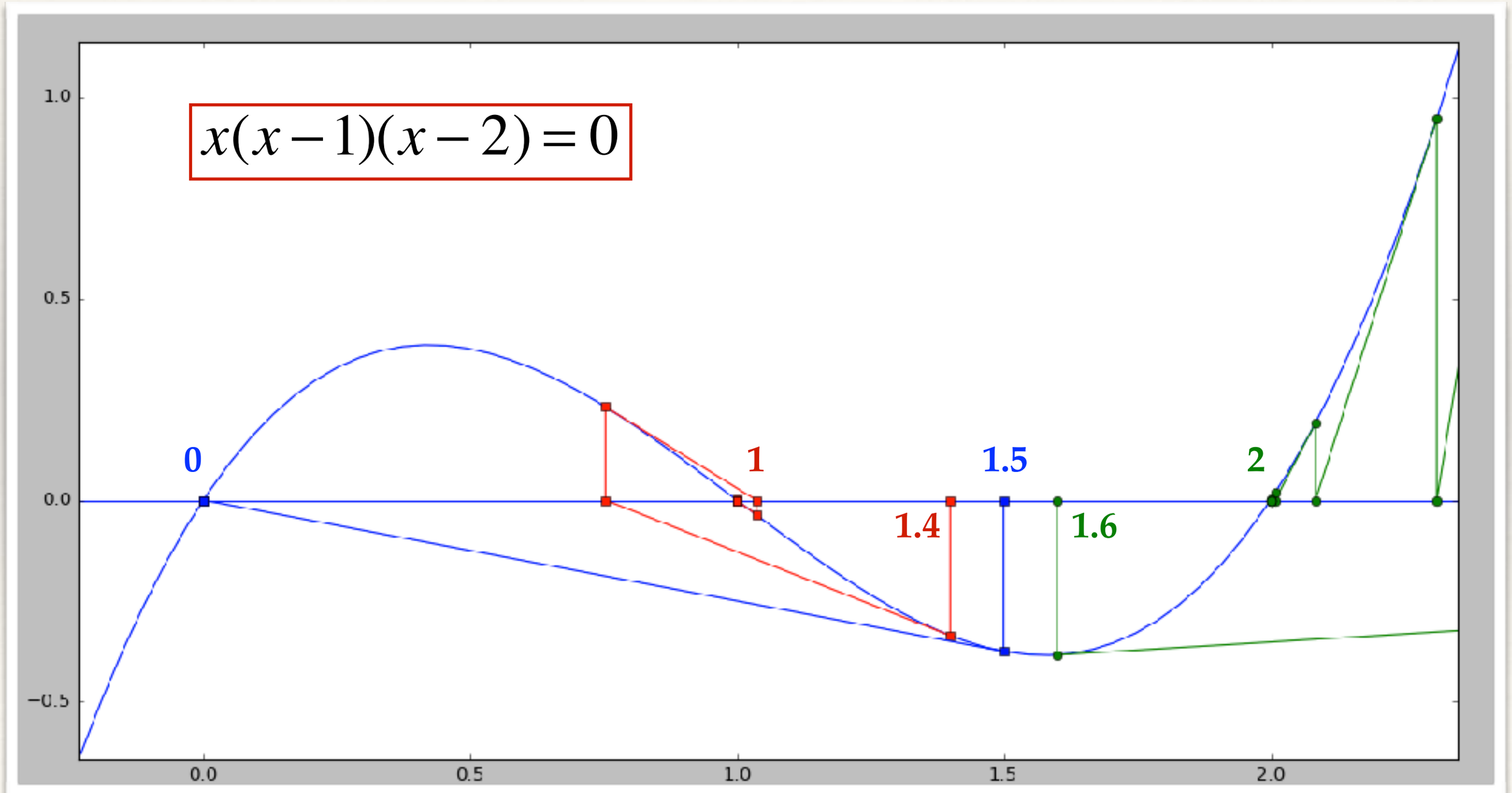


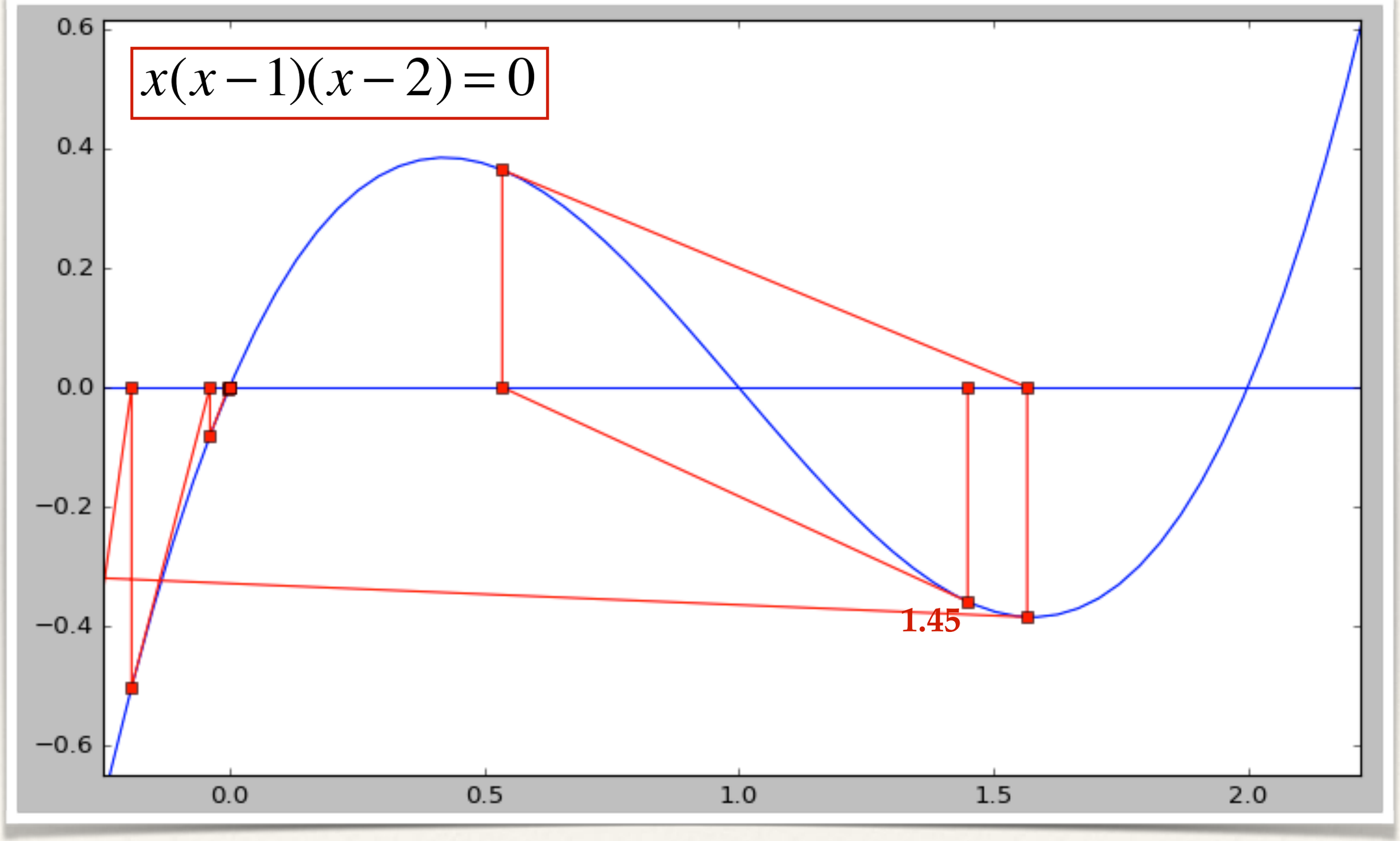


La méthode ne fonctionne pas pour une condition initiale en 0. (minimum local) et le temps de calcul diverge quand on s'en rapproche.

Exemples de cas plus problématiques

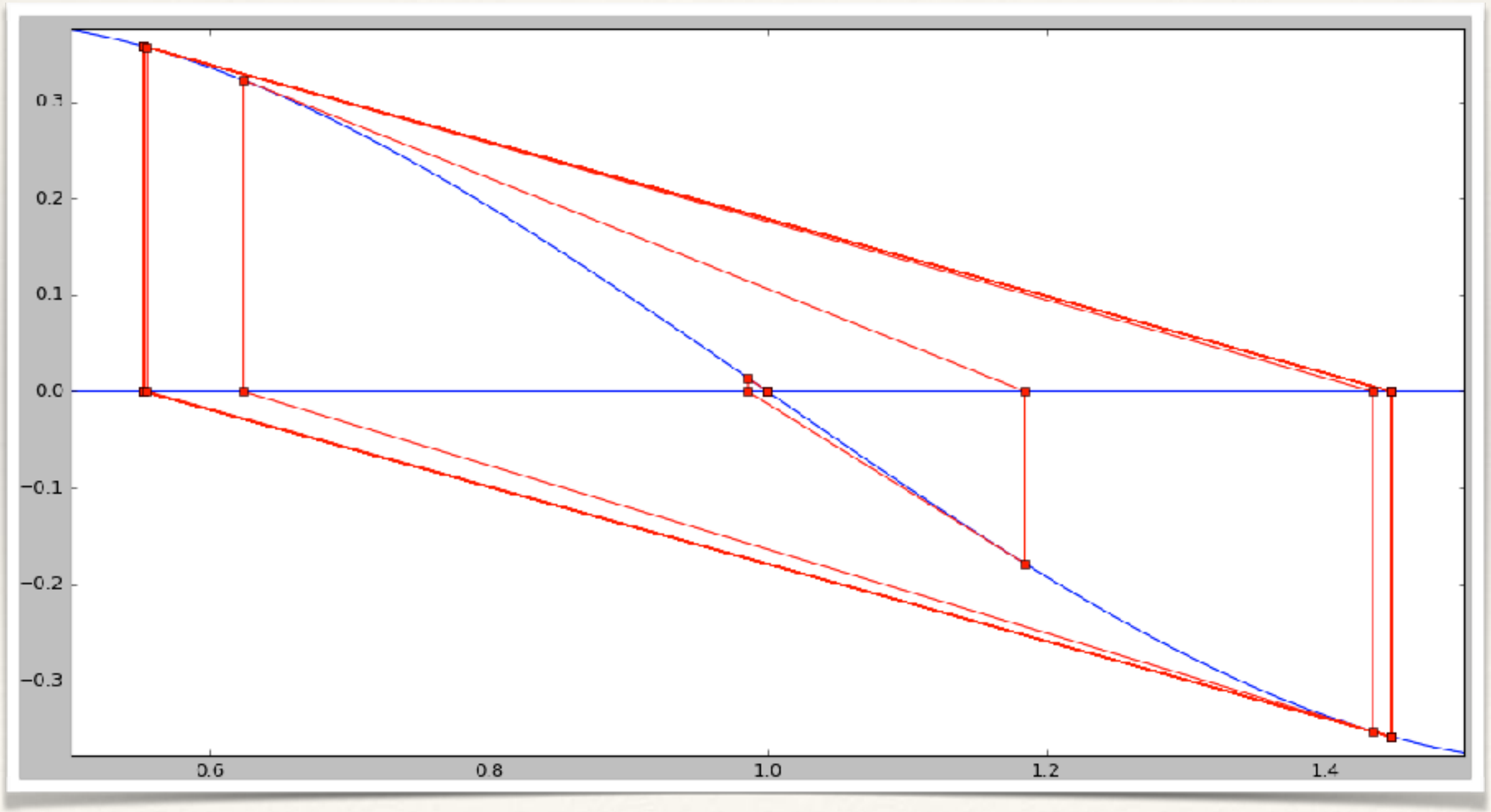
Il peut rapidement y avoir une forte instabilité sur les conditions initiales, même pour des courbes très régulières :

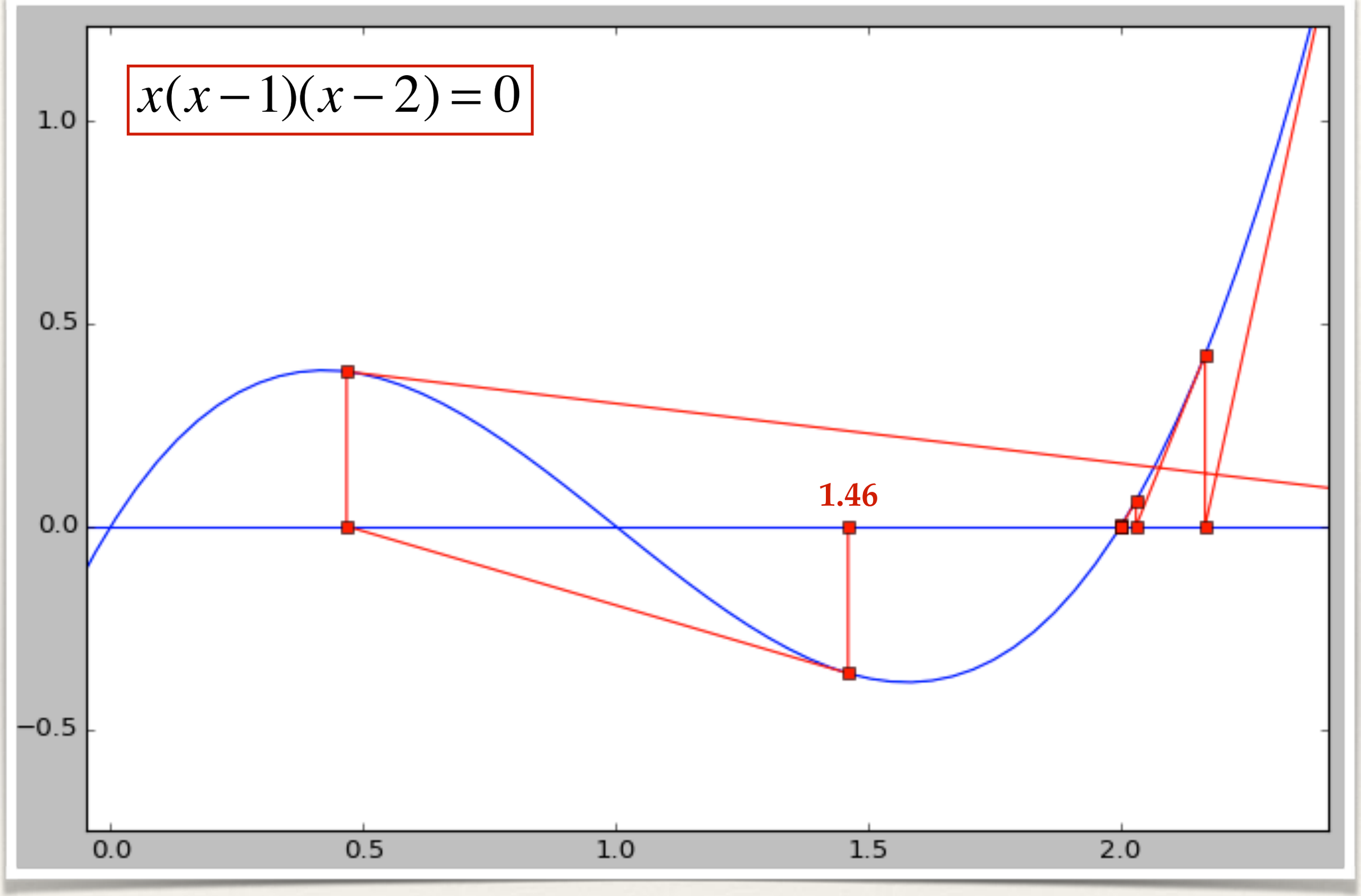



$x(x-1)(x-2)=0$
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.5
1.0
1.5
2.0
1.45

Dans de rares cas on peut boucler indéfiniment sur les mêmes positions …

$$X_0 = 1.44721359549995798321 3$$





Il n'est pas possible de deviner le bon choix pour les conditions initiales dans un cas général.