TP 1 : Optimisation en dimension un — Éléments de correction

```
In [1]: %pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

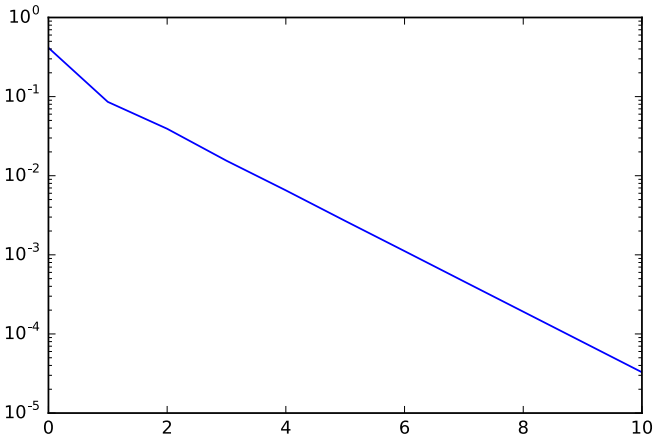
1. Ordre de convergence des suites

On peut tracer $|x_n - x_\infty|$ de deux manières : soit en connaissant la valeur théorique $\sqrt{2}$, soit en approximant x_∞ par le dernier point calculé. On peut utiliser la fonction `abs` qui s'applique sur des tableaux de type array (donc il faut transformer la liste en array si on a généré la suite par une liste, on peut aussi directement remplir un tableau vu qu'on connait sa taille a priori).

On doit observer une droite en échelle semi-logarithmique pour le cas (a), et des courbes ressemblant à l'opposé d'une exponentielle dans les cas (b) et (c). Pour mieux le voir, on peut tracer $|\ln|x_n - x_\infty||$ en échelle semi-log et voir des droites. Mais elles sont tronquées du fait que l'on atteint rapidement la précision machine (5 itérations pour (b), 7 pour (c)).

Voici plusieurs exemples (traités de manière différentes pour chaque suite) de ce qui pouvait être attendu.

```
In [2]: x=1
        liste=[x]
        for i in range(10):
            x=x*(1-x/2)+1
            liste.append(x)
        semilogy(abs(array(liste)-sqrt(2)))
        show()
```

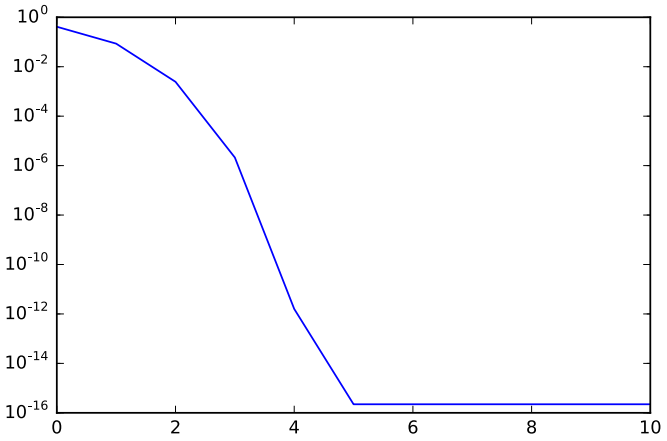


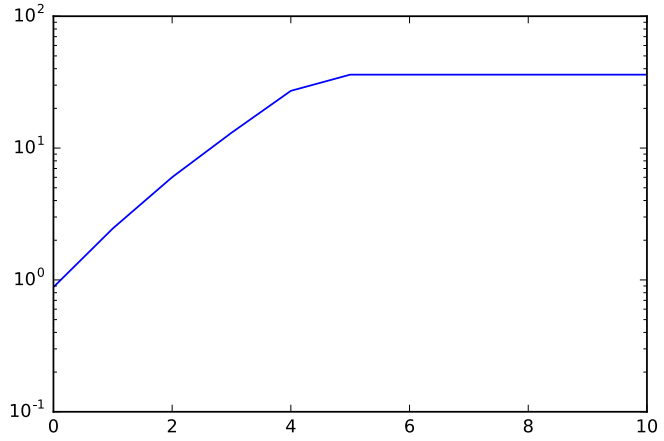
On obtient bien une droite de pente négative. Cela indique que $\ln(|x_n - x_\infty|)$ se comporte bien, lorsque n est grand, comme $C - an$, et donc que $|x_n - x_\infty|$ a un comportement du type $C'\alpha^n$ (avec $\alpha = e^{-a} < 1$).

L'échelle logarithmique permet de voir que l'on gagne des décimales à un taux constant : il faut autant d'itérations pour passer de 10^{-2} à 10^{-3} que de 10^{-3} à 10^{-4}. C'est en cela qu'on dit que la convergence est linéaire.

```
In [3]: x=1
        tableau=zeros(11)
        tableau[0]=x
        for i in range(1,11):
            x=x/2+1/x
            tableau[i]=x
```

```
erreurs=abs(tableau-sqrt(2))
semilogy(erreurs)
show()
semilogy(abs(log(erreurs)))
show()
```





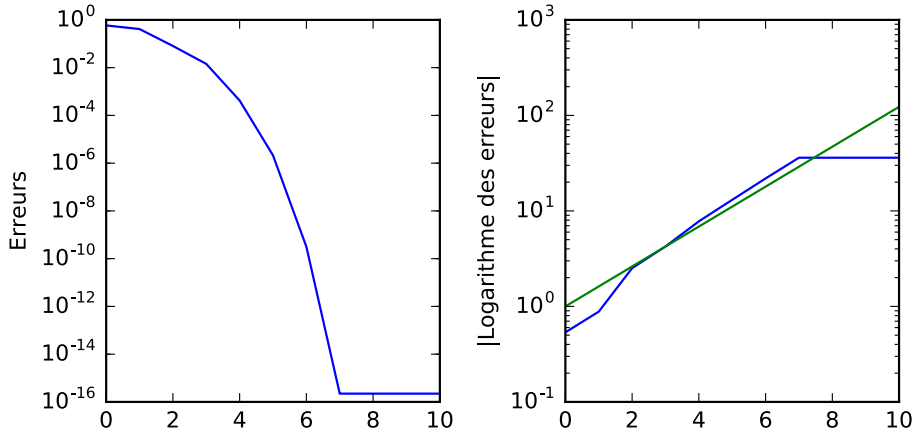
On voit bien que l'erreur atteint la précision machine (de l'ordre de 10^{-16}) au bout de 5 itérations. On voit sur la première courbe que la décroissance de l'erreur sur les premières itérations est bien plus rapide que si c'était une convergence linéaire : on gagne de plus en plus de décimales à chaque itération.

Sur la deuxième courbe, pour ces premières itérations on observe quelque chose qui ressemble à une droite de pente positive. Comme on a tracé le logarithme de l'erreur, et qu'on l'a affiché en échelle semi-logarithmique pour les ordonnées, cela indique que $|\ln(|x_n - x_\infty|)|$ se comporte en fait comme $C\beta^n$, cette fois-ci avec un $\beta > 1$ (puisque la pente est positive). Cela indique que $|x_n - x_\infty|$ se comporte comme α^{β^n} avec $\alpha = e^{-C} < 1$.

Il est difficile de bien observer le β vu qu'on a peu d'estimation, mais on peut voir sur le premier graphique que le nombre de décimale double environ à chaque itération : on passe d'un peu plus de 10^{-3} à 10^{-6} puis 10^{-12} aux itérations 2, 3 et 4. C'est le signe que la convergence est quadratique.

```
In [4]: # Sans utiliser x, seulement la valeur dans le tableau
        tableau=zeros(11)
        tableau[0],tableau[1]=2,1
```

```
for i in range(1,10):
    tableau[i+1]=(tableau[i]*tableau[i-1]+2)/(tableau[i]+tableau[i-1])
erreurs=abs(tableau-sqrt(2))
phi=(1+sqrt(5))/2
# Exemple d'affichage de deux graphiques côte à côte
figure(figsize=(6,3)) # Pour gérer la taille totale de la figure
subplot(1,2,1)
ylabel("Erreurs")
semilogy(erreurs)
subplot(1,2,2)
ylabel("|Logarithme des erreurs|")
semilogy(abs(log(erreurs)))
semilogy(phi**(arange(11)))
tight_layout()  # Pour que légendes et graphiques ne se chevauchent pas
show()
```



On a les mêmes remarques que précédemment. Comme on a un peu plus de points, on a tracé tracé les points d'ordonnées φ^n en échelle logarithmique sur le graphique de droite, ce qui donne une droite de pente correspondant approximativement à celle observée sur la courbe des erreurs. Ceci indique que l'ordre est effectivement φ.

2. Dichotomie et dérivée approchée.

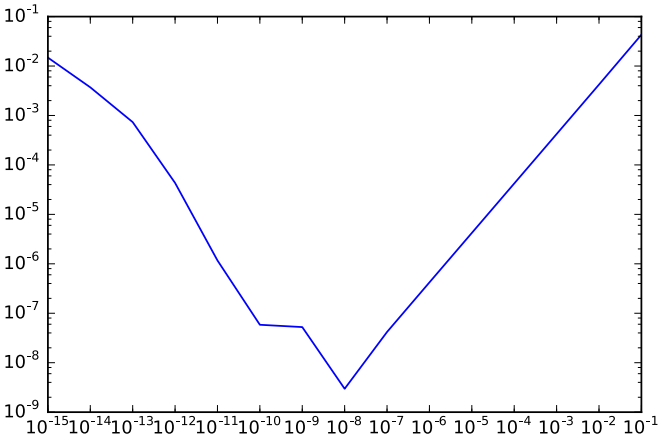
(a) Deux manières d'intégrer le ε : soit comme variable globale à l'extérieur de la fonction de calcul de dérivée, soit comme paramètre.

Attention à la plage des ε que l'on veut tester : si on utilise `linspace` entre 10^{-15} et 10^{-1}, on obtient des ε qui ont tous le même ordre de grandeur (à part le premier qui vaut 10^{-15}, le suivant est déjà de l'ordre de $\frac{1}{N}10^{-1}$, où N est le nombre de points). Deux solutions à cela : soit faire une boucle sur des entiers i allant de -15 à -1 et prendre $\varepsilon = 10^{-i}$, soit utiliser une commande du genre `10**linspace(-15,-1)`, qui est en fait connue comme la commande `logspace(-15,-1)`. Suivant que l'on affiche les i ou directement les ε en abscisses, il faudra ou non prendre une échelle logarithmique sur l'axe des abscisses : utiliser `semilogy` ou `loglog`. En effet, on aura toujours intérêt à afficher une échelle logarithmique sur l'axe des ordonnées, pour observer l'ordre de grandeur de l'erreur obtenue.

On doit observer que le bon choix se situe aux alentours de $\varepsilon = 10^{-8}$, et l'erreur est du même ordre. L'explication à donner est que l'erreur relative d'arrondi lors du calcul de $x + \varepsilon$ est plus forte si ε est trop petit, et l'erreur de la méthode d'approximation est forte si ε est trop gros.

```
In [5]: def diffFinies(f,eps,x):
            return (f(x+eps)-f(x))/eps
```

```
In [6]: erreurs=zeros(15)
        epsilons=[10**(-i) for i in range(1,16)]
        for i in range(15):
            erreurs[i]=abs(diffFinies(sin,epsilons[i],1)-cos(1))
        loglog(epsilons,erreurs);show()
```



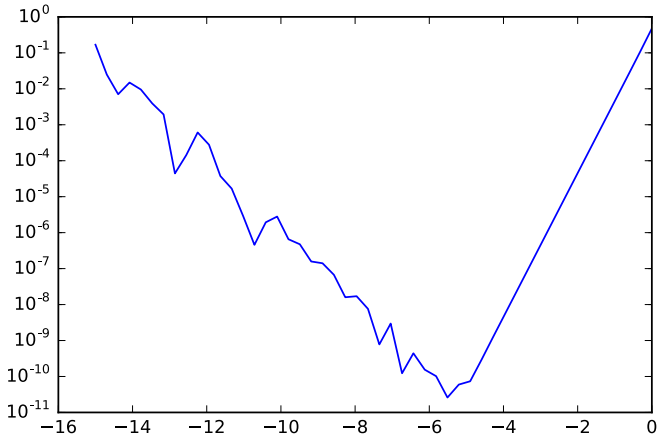
On observe bien que la valeur de l'erreur minimale est de l'ordre de 10^{-8} lorsque ε est aussi de l'ordre de 10^{-8}.

(b) On doit cette fois-ci obtenir un bon choix aux alentours de $\varepsilon = 10^{-5}$ et l'erreur est de l'ordre de 10^{-10} environ (cela peut dépendre légèrement des exemple de fonctions que l'on prend et les point où on les évalue.

Attention ici, si l'on prend comme fonction la fonction carré (choix souvent fait par les élèves), on n'observe pas d'erreur lorsque ε est grand : c'est normal, car la différence finie centrée est alors une formule exacte. De même si on prend le sinus en 0 par exemple, la dérivée troisième s'annulant en 0, on obtient une meilleure approximation théorique, et donc le bon choix n'est pas exactement du même ordre.

```
In [7]: epsilon=0.1 # Version avec une variable globale
        def diffCentrees(f,x):
            return (f(x+epsilon)-f(x-epsilon))/(2*epsilon)
```

```
In [8]: listerreurs=[]
        for i in linspace(-15,0):
            epsilon=10**(i)
            listerreurs.append(abs(diffCentrees(exp,1)-exp(1)))
        semilogy(linspace(-15,0),listerreurs);show()
```



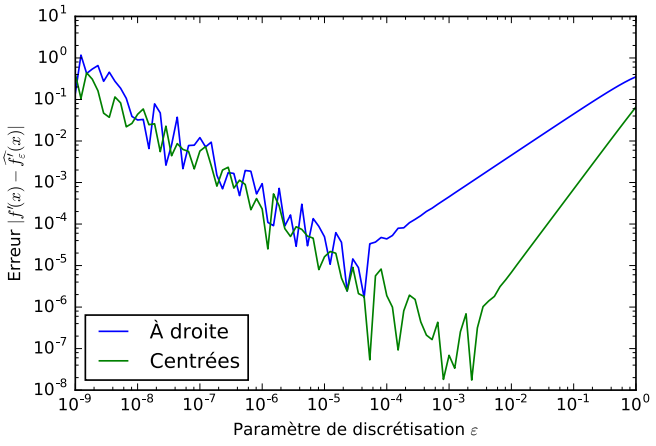
L'erreur est bien minimale quand ε est de l'ordre de 10^{-5}, et elle est bien de l'ordre de 10^{-10}. On observe que du côté où l'erreur numérique (dûe au arrondis) domine (quand $\varepsilon \ll 10^{-5}$), le comportement de l'erreur est un peu aléatoire, alors que du côté où c'est l'erreur théorique qui domine, elle se comporte linéairement avec la taille de ε.

(c) Le bon choix doit être de l'ordre de grandeur de la racine de `tailleErreur` pour la différence finie simple, et de la racine cubique pour la différence finie centrée (d'après les résultats qu'on obtient ensuite théoriquement aux questions (d) et (e)).

```
In [9]: tailleErreur=1e-9
        def sinapprox(x):
            return (2*random_sample()-1)*tailleErreur + sin(x)

        epsilons=logspace(-9,0,100)
        erreurs1=[abs(diffFinies(sinapprox,eps,2)-cos(2)) for eps in epsilons]
        loglog(epsilons,erreurs1)

        erreurs2=zeros_like(epsilons) # un array de même taille que epsilons
        for (i,eps) in enumerate(epsilons): # indices et valeurs en même temps
            epsilon=eps
            erreurs2[i]=abs(diffCentrees(sinapprox,2)-cos(2))
        loglog(epsilons,erreurs2)
        legend(['À droite',"Centrées"],loc='lower left')
        # loc="..." permet de localiser la légende dans un coin
        xlabel('Paramètre de discrétisation $\\varepsilon$')
        ylabel("Erreur $|f'(x)-\\widehat{f_{\\varepsilon}}'}(x)|$")
        show()
```



On a bien une erreur minimale d'ordre entre 10^{-4} et 10^{-5} pour un ε du même ordre pour les différences finies à droite, et une erreur d'ordre 10^{-6} pour un ε de l'ordre de 10^{-3}, lorsque la variable `tailleErreur` (qui correspond au η des questions suivantes) vaut 10^{-9} (qui s'écrit `1e-9` en notation de nombre à virgule flottante). Remarquer aussi qu'on peut mettre du LaTeX dans les décorations des graphiques en remplaçant les \ par \\.

(d) Voilà un exemple extrêmement minimaliste. Pour l'améliorer, on pourrait mettre une option qui calcule par différences centrées au lieu de différences finies, ou ajouter des tests pour s'assurer qu'on a donné des points initiaux corrects. Par exemple on pourra écrire `assert(b>a)` pour renvoyer une erreur si on ne rentre pas les points ordonnés.

```
In [10]: def minDicho(f,a,b,eps,tol):
             while abs(b-a)>tol:
                 c=(a+b)/2
                 fpc=diffFinies(f,eps,c)
                 if fpc>0:
                     b=c
                 else:
                     a=c
             return c
```

(e) Attention ici, si on veut modifier la valeur d'une variable globale `n` à l'intérieur d'une fonction, il faut utiliser l'instruction `global n` avant son utilisation dans la fonction. Si on utilise simplement une variable globale, on n'a pas besoin de cette instruction, mais si l'on veut la modifier, il faut le préciser.

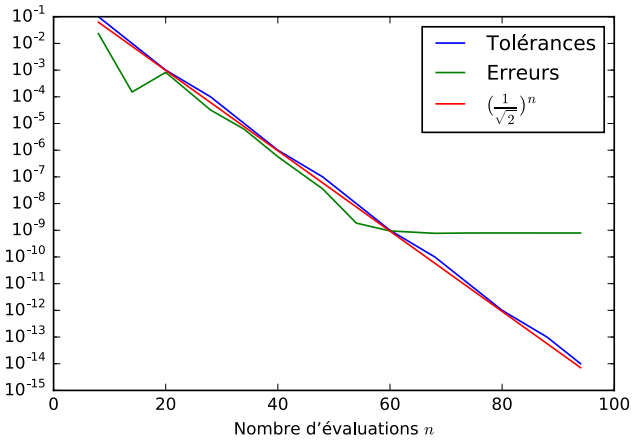
Pour tracer le taux de convergence effectif, une manière est de se donner une liste de tolérances, et de calculer le nombre d'évaluations de fonctions (à l'aide du compteur) nécéssaires à ce que la boucle s'arrête. On peut alors tracer en abscisses le nombre d'évaluations n et en ordonnées la tolérance (ou l'erreur entre le point obtenu x_k et le minimum x_∞, ou une approximation de x_∞ par le point x_k obtenu avec la plus petite tolérance). On prendra une échelle logarithmique en ordonnée pour voir que l'erreur semble se comporter comme α^n.

On peut par exemple tracer en plus une droite de pente $\frac{1}{\sqrt{2}}$ en échelle logarithmique : on trace les points d'ordonnée $(\frac{1}{\sqrt{2}})^n$ (avec n en abscisse). En échelle logarithmique pour les ordonnées on doit observer une droite « parallèle » à l'ensemble des points précédents.

```
In [11]: compteur=0
         def f(x):
             global compteur
             compteur += 1
             return x/2+1/x
```

```
In [12]: tolerances=[10**(-i) for i in range(1,15)]
         nevaluations=[]
         erreurs=[]
         eps=1e-8 # Le meilleur choix d'après (a) : différences finies à droite.
         for tol in tolerances:
             compteur=0
             c=minDicho(f,1,2,eps,tol)
             erreurs.append(abs(sqrt(2)-c))
             nevaluations.append(compteur)
         semilogy(nevaluations,tolerances)
         semilogy(nevaluations,erreurs)

         semilogy(nevaluations,(1/sqrt(2))**nevaluations)
         legend(['Tolérances','Erreurs','$(\\frac{1}{\\sqrt{2}})^n$'])
         xlabel('Nombre d'évaluations $n$')
         show()
```



On observe donc bien un taux de convergence effectif de $\frac{1}{\sqrt{2}}$. Le fait que les erreurs se stabilisent autour de 10^{-9} est simplement dû au fait qu'on ne peut pas avoir une meilleure précision avec des différences finies centrées.

3. Méthodes de réduction de triplets

(a). L'algorithme est donné dans le cours, la seule chose à faire est d'ajouter un critère d'arrêt lorsque la taille du segment est plus petite que ε.

Il faut bien vérifier que le nombre d'évaluations par itération est bien 1. Pour cela, bien faire attention à garder en mémoire les valeurs de $f(a)$, $f(b)$, $f(c)$ et $f(d)$ pour ne pas les recalculer, par exemple ne pas écrire `if f(c)<f(d)`, car cela calculera deux fois f, alors que le calcul avait déjà été fait. Rajouter par exemple des variables `fa`, `fb`, `fc`, `fd` qui gardent en mémoire l'évaluation de f aux points a_n, b_n, c_n, d_n. En pratique, on n'a même pas besoin de garder en mémoire $f(a)$ et $f(b)$: on ne fait de comparaison que sur $f(c)$ et $f(d)$. Si on veut être plus précautionneux, on peut rajouter un calcul de $f(a)$ et $f(b)$ au départ pour s'assurer qu'on part bien d'un premier triplet admissible, et renvoyer une erreur sinon...

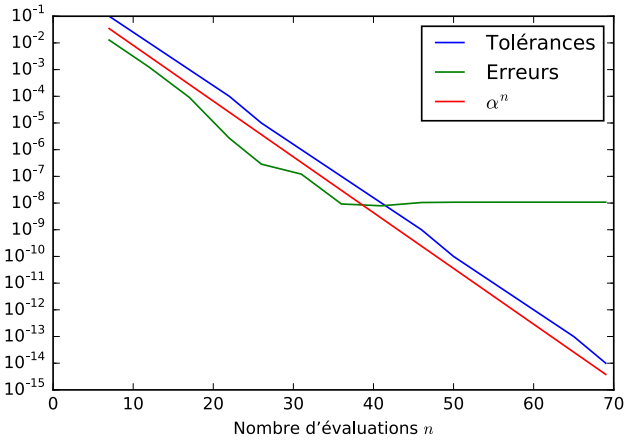
Au moment de la mise à jour d'un triplet, par exemple pour faire $(a_{n+1}, d_{n+1}, b_{n+1}) = (a_n, c_n, d_n)$, on pourra utiliser l'affectation simultanée : plutôt que de faire une variable temporaire `tmp` et d'écrire `tmp=d`, `d=c`, `b=tmp` (qu'on aurait pu éviter en changeant l'ordre et écrivant `b=d` puis `d=c`), on peut directement écrire `d,b=c,d`, ce qui facilite la lecture.

```
In [13]: alpha=(sqrt(5)-1)/2
         def minDoree(f,a,b,tol):
             c=a+(1-alpha)*(b-a)
             d=a+alpha*(b-a)
             fc,fd=f(c),f(d)
             while b-a>tol:
                 if fc>fd:
                     a,c=c,d
                     d=a+alpha*(b-a)
                     fc,fd=fd,f(d)
                 else:
                     d,b=c,d
                     c=a+(1-alpha)*(b-a)
                     fc,fd=f(c),fc
             return (a+b)/2
```

(b). Comme précédemment (cf. (g) de la partie 2), il faut utiliser le mot-clé `global` pour pouvoir modifier la variable n à l'intérieur de la fonction. Mais on peut garder la même fonction. On peut observer le taux de la même manière.

```
In [14]: tolerances=[10**(-i) for i in range(1,15)]
         nevaluations=[]
         erreurs=[]
         for tol in tolerances:
             compteur=0
             c=minDoree(f,1,2,tol)
             erreurs.append(abs(sqrt(2)-c))
             nevaluations.append(compteur)
         semilogy(nevaluations,tolerances)
         semilogy(nevaluations,erreurs)

         semilogy(nevaluations,alpha**nevaluations)
         legend(['Tolérances','Erreurs','$\\alpha^n$'])
         xlabel('Nombre d'évaluations $n$')
         show()
```



On observe bien un taux de convergence effectif d'ordre α. Comment expliquer que les erreurs par rapport à la vraie valeur du minimum se stabilisent autour de 10^{-8} ? C'est un point sur lequel je vous laisse méditer... Et si vous venez me poser une question sur ça, c'est au moins la preuve que vous avez lu jusqu'ici, ce qui me réjouis dans tous les cas.

(c*) L'algorithme est donné dans le cours. Le critère d'arrêt pour une tolérance ε est par exemple lorsque l'espacement entre les trois derniers points est plus petit que ε : $\max(x_{k+1}, x_k, x_{k-1}) - \min(x_{k+1}, x_k, x_{k-1}) \leqslant \varepsilon$. Mais en pratique choisir comme critère d'arrêt $|x_{k+1} - x_k| \leqslant \varepsilon$ est suffisant.

4. Application : résolution d'un problème de plus court chemin entre deux zones parcourues à deux vitesses différentes.

(a) Pour prendre les points au hasard, on peut tirer $x_a \in \mathbb{R}$ selon une loi normale avec `randn`, puis tirer un nombre positif correspondant à $y_a - f(x_a)$ pour être sûr d'être dans la zone Z_1. De même pour x_b et $f(x_b) - y_b$.

Pour faire un joli graphique, on pourra utiliser la fonction `fill_between` pour colorer une des deux zones.

Pour ce qui est du choix de la fonction convexe, vous en connaissez des tonnes : exp, $x \mapsto x^2$, $x \mapsto \sqrt{1+x^2}$ $x \mapsto ax+b$, ainsi que toute combinaison convexe de ces fonctions. On a aussi $x \mapsto 1/x$, $-\ln$ (qui ne s'appliquent pas forcément dans ce cadre théorique, mais en pratique on se place sur des intervalles où elles sont bien définies et cela fonctionne).

```
In [15]: def f(x):
             return log(1+exp(3*x))/2-x+1
```

```
xa=4*rand()-2 # entre -2 et 2
xb=4*rand()-2
yb=f(xb)-2*rand()
ya=f(xa)+2*rand()
```

In [16]:
```
v1,v2=2,1

def F(x):
    fx=f(x)
    return norm([x-xa,fx-ya])/v1+norm([x-xb,fx-yb])/v2

xm=minDoree(F,-2,2,1e-4) # On ne cherche pas plus précis
                         # Ça ne se verrait pas sur le graphique...
```

On a intérêt de faire une nouvelle cellule pour afficher le graphique : on aura besoin de peaufiner les zones pour que ce soit joli, et il n'est pas question de relancer le code ci-dessus, qui tirerait de nouveau des points A et B aléatoirement...

In [17]:
```
X=linspace(-2,2)
plot(X,f(X))
fill_between(X,f(X),0,alpha=0.3)
plot([xa,xm,xb],[ya,f(xm),yb],'-ro')
text(xa,ya+.1,"$A$",color='r',horizontalalignment='center')
text(xb,yb-.2,"$B$",color='r',horizontalalignment='center')
text(xm+.1,f(xm)+.2,"$M_*$",color='r',horizontalalignment='center')
text(0,.2,"$Z_2$ : vitesse $v_2$",color='b',horizontalalignment='center')
text(0,3,"$Z_1$ : vitesse $v_1$",horizontalalignment='center')
axis('scaled')
show()
```