L'approximation des intégrales

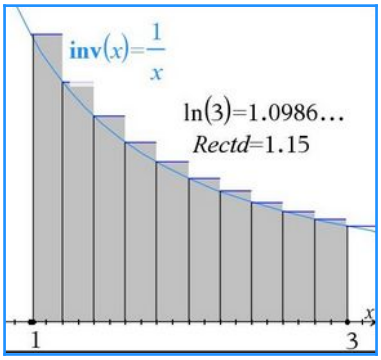
Comment « calculer » une intégrale ?

Dans les programmes

Contenus Définition de l'intégrale d'une fonction continue positive définie sur un segment [a, b], comme aire sous la courbe représentative de f. [Lien entre intégrales et primitives]	**Capacités attendues** Estimer graphiquement ou encadrer une intégrale. Calculer l'aire entre deux courbes. **Approfondissements** Approximation d'une aire par l'utilisation de suites adjacentes. **Exemples d'algorithme** Méthodes des rectangles, des milieux, des trapèzes.

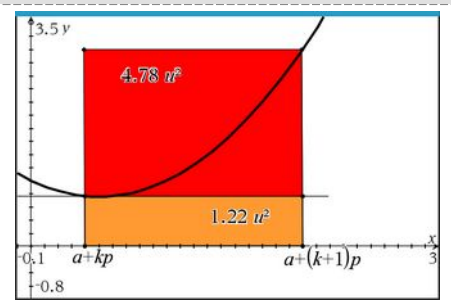
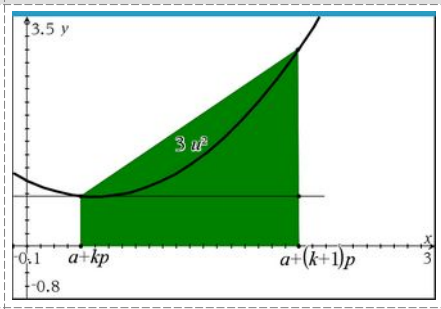
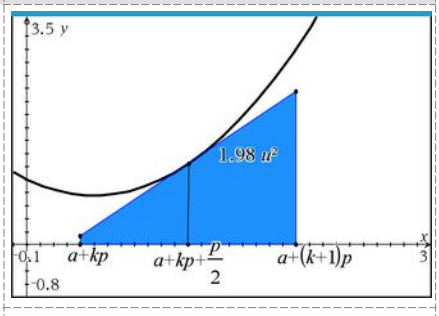
Situation déclenchante

– *(l'élève) Madame, peut-on toujours calculer des intégrales avec des formules comme pour les dérivées ?*
– *Non, ce n'est pas toujours possible. Le mathématicien Liouville l'a démontré en 1835.*
– *Alors, on fait comment ?*
– *Si tu acceptes une petite erreur d'approximation, on peut y parvenir.*
– *D'accord. Comment ça se passe ?*
– *Tu pourrais déjà découper ton intégrale en « bandelettes » très fines, dont l'aire est bien proche de celle d'un rectangle …*
– *Ah ! Mais je peux même faire un peu mieux !*



Il s'agit de déterminer une valeur approchée de l'intégrale d'une fonction f continue et positive sur un intervalle $[a\,;b]$ grâce aux algorithmes des rectangles, milieux ou trapèzes. Pour approcher $\int_a^b f(x)\,\mathrm{d}x$ on commence par la découper en n sous-intégrales sur des intervalles successifs, chacun de longueur $p=\frac{b-a}{n}$ (relation de Chasles, voir ci-contre). Les « points » du découpage sont $x_k=a+k\times p$ pour $0\leq k\leq n$.

On approche cette dernière par l'aire d'un rectangle ou d'un trapèze, en suivant l'une des méthodes indiquées ci-dessous.

Rectangles	Trapèzes	Point milieu
		
$\text{RecG}=p\sum_{k=0}^{n-1} f(x_k)$ $\text{RecD}=p\sum_{k=1}^{n} f(x_k)$	$\text{Trap}=\frac{p}{2}\sum_{k=0}^{n-1}\left[f(x_k)+f(x_{k+1})\right]$	$\text{Mil}=p\sum_{k=0}^{n-1} f\left(a+k\times p+\frac{p}{2}\right)$

L. DIDIER & R. CABANE

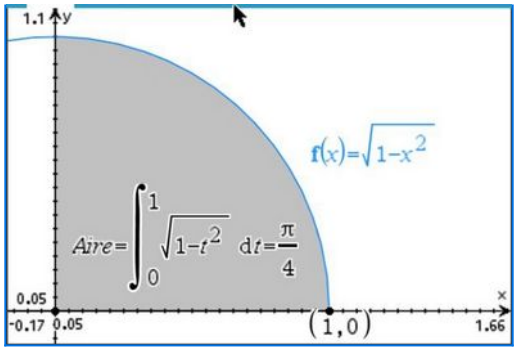
Objectifs

▶ Objectif 1 : programmer le calcul approché, travail de groupe

La classe se répartit en plusieurs petits groupes, qui choisissent l'un des quatre algorithmes cités, et les programment en langage Python. À la suite, les groupes testent leurs programmes pour le calcul de $\ln(2) \approx 0.6931472$ (comme intégrale de la fonction inverse définie sur l'intervalle $[1\,;\,2]$ par $\text{inv}(x)=\frac{1}{x}$) et comparent leurs résultats : valeurs par excès ou par défaut, avec quel écart ?

Validation

Pour tester nos algorithmes, nous pouvons tenter d'approcher « au mieux » $I=\int_0^1 \sqrt{1-x^2}\,\mathrm{d}x$. Cette intégrale est bien connue[1] : elle mesure l'aire d'un quart de disque de rayon 1, valant $\frac{\pi}{4}$ unités d'aire (voir la figure ci-contre[2]).



▶ Objectif 2 : les primitives

Dernier objectif : tracer la courbe représentative d'une primitive F d'une fonction f définie et continue sur un certain intervalle. On pourrait évidemment approcher les valeurs de la primitive au moyen du travail antérieur, mais c'est peu efficace car on recalculerait de nombreuses fois les mêmes valeurs. Mieux vaut donc procéder « de proche en proche » : une fois $F(x)$ approchée, on peut approcher $F(x+h)$ (h étant le « pas », une petite valeur) à partir de la formule $F(x+h)=F(x)+\int_x^{x+h} f(t)\,\mathrm{d}t$, où l'intégrale peut être estimée par l'une des méthodes précédentes.

On propose donc d'écrire une fonction ayant pour appel : `prim(f,a,b,h,y0)`, `f` étant la fonction à primitiver, `a` et `b` les bornes, `h` le « pas » et `y0` la valeur de la primitive au point a, et renvoyant une liste formée de deux listes, celle des abscisses utilisées et celle des ordonnées correspondantes pour F.

En d'autres termes, il s'agit de calculer, de proche en proche, les valeurs de $F(x)=y_0+\int_a^x f(t)\,\mathrm{d}t$.

Et un exemple à traiter : le logarithme bien sûr (en partant de $\ln(1)=0$).

Attention : la fonction « logarithme népérien » (notée **ln**) est codée sous le nom `log` en Python !

1 La primitive de la fonction $x \mapsto \sqrt{1-x^2}$ a une expression fort compliquée.

2 Cette figure, comme les précédentes, a été réalisée avec le logiciel TI-Nspire CX CAS.

L. DIDIER & R. CABANE

Une fonction = un objet

Dans nos fonctions d'approximation `recg`, `recd`, `mil`, `trap`, on a quatre paramètres : la fonction *f*, les bornes *a*, *b* et le nombre *n* de sous-intervalles à la base du découpage.

Notons qu'une fonction peut être passée comme paramètre à une autre fonction.
L'exemple ci-contre confirme cette possibilité.

Ici `carre` désigne une fonction Python.

```
ÉDITEUR : EVALUE
LIGNE DU SCRIPT 0006
def evalue1(f,x):
  return f(x+1)-f(x)

def carre(x):
  return x*x
```

```
PYTHON SHELL
>>> evalue1(carre,2)
5
>>> evalue1(exp,0)
1.718281828459046
```

Objectif 1 : programmes en Python

La programmation des approximations en rectangles à gauche (`recg`) et à droite (`recd`) suit exactement les prescriptions antérieures. On définit deux variables internes (locales) à la fonction, `s` pour accumuler les valeurs de la fonction `f` et `p` pour la largeur des intervalles (le « pas »).

Pour les trapèzes, on évite de transcrire directement le calcul des aires des trapèzes, soit $p\times\frac{f(x_k)+f(x_{k+1})}{2}$, car cela conduirait à calculer deux fois les valeurs de la fonction f excepté les valeurs extrêmes $f(a)$ et $f(b)$.

On garde ainsi le même schéma d'accumulation de valeurs dans une variable `s`, mais on initialise `s` avec les valeurs extrêmes.

On utilise ici une variante du mécanisme `range` : `range(1,n)` fait prendre à `k` les valeurs entières de 1 à `n-1` (inclus).

Pour le point milieu, on décale simplement le point d'évaluation de f d'une demi-longueur $\frac{p}{2}$.

```
ÉDITEUR : RECTANGL
LIGNE DU SCRIPT 0013
def recg(f,a,b,n):
  s=0
  p=(b-a)/n
  for k in range(n):
    s=s+f(a+k*p)
  return s*p

def recd(f,a,b,n):
  s=0
  p=(b-a)/n
  for k in range(n):
    s=s+f(a+k*p+p)
  return s*p

def trap(f,a,b,n):
  s=(f(a)+f(b))/2
  p=(b-a)/n
  for k in range(1,n):
    s=s+f(a+k*p)
  return s*p

def mil(f,a,b,n):
  s=0
  p=(b-a)/n
  for k in range(n):
    s=s+f(a+k*p+p/2)
  return s*p

def f(x): return sqrt(1-x*x)

def inv(x): return 1/x
```

Validations

On commence par intégrer la fonction « inverse » (`inv`) entre 1 et 2, en vue d'approcher $\ln(2)$; on voit de suite que pour un même nombre d'intervalles c'est l'approximation des milieux qui semble la meilleure. Cela dit, pour réaliser une approximation conséquente il faudrait augmenter substantiellement le nombre d'intervalles, au prix d'un temps de calcul considérable.

Pour l'aire du quart de disque, même constat : un meilleur algorithme peut conduire à de meilleures approximations, mais nous sommes encore loin de pouvoir atteindre des précisions de l'ordre de 10^{-9} ou mieux !

```
PYTHON SHELL
>>> recg(inv,1,2,100)-log(2)
0.002506249921878867
>>> recd(inv,1,2,100)-log(2)
-0.002493750078121137
>>> trap(inv,1,2,100)-log(2)
6.249921878809239e-06
>>> mil(inv,1,2,100)-log(2)
-3.1249316446713l4e-06
>>> mil(inv,1,2,1000)-log(2)
-3.124999337078549e-08
>>> recg(f,0,1,100)
0.7901042579447612
>>> recd(f,0,1,100)
0.7801042579447612
>>> trap(f,0,1,100)
0.7851042579447612
>>> mil(f,0,1,100)
0.7854842144750021
```

Objectif 2 : primitives

Nous allons ici employer la méthode des milieux. En vue du tracé de courbe, il est plus simple de « remplir » deux listes (ici LX et LY), l'une avec les abscisses et l'autre avec les ordonnées des points de la courbe représentative de la primitive.

On initialise le processus avec la valeur attendue pour la primitive au point a, soit ici `y0`.

TI-83 La mise en œuvre graphique nécessite de faire appel au module `ti_plotlib` (ne fonctionne pas sur la TI-82). Pour comparer avec la « courbe » représentative du logarithme, on crée dans la liste Z les ordonnées correspondantes aux mêmes abscisses. Le reste n'est plus que mise en place graphique.

Les résultats, ci-dessous, montrent que l'approximation s'approche assez correctement de la courbe du logarithme. Compte tenu de la mémoire disponible sur la TI-83, il est impossible de faire mieux que p=0,045.

```
ÉDITEUR : RECTANGL
LIGNE DU SCRIPT 0035
def prim(f,a,b,p,y0):
  LX=[a]
  LY=[y0]
  x=a
  y=y0
  while x<b:
    x=x+p
    y=y+f(x+p/2)*p
    LX.append(x)
    LY.append(y)
  return [LX,LY]
Fns… | a A # | Outils | Exéc | Script
```

La fonction `prim()` renvoie un couple (une liste) formé de deux listes (abscisses, ordonnées).

```
ÉDITEUR : RECTANGL
LIGNE DU SCRIPT 0048
from ti_plotlib import *

def d(p):
  [X,Y]=prim(inv,1,5,p,0)
  Z = [log(x) for x in X]
  cls()
  window(1,5,0,2)
  text_at(1,"p="+str(p),"left")
  color(0,192,0)
  grid(1,.5)
  color(0,0,255)
  text_at(1,"p="+str(p),"left")
  color(0,192,0)
  grid(1,.5)
  color(0,0,255)
  text_at(5,"ln","center")
  color(255,0,0)
  text_at(4,"prim","right")
  plot(X,Y,".")
  color(0,0,255)
  plot(X,Z,".")
  show_plot()
Fns… | a A # | Outils | Exéc | Script
```

Le résultat de la fonction `prim()` est rangé dans une liste formée de deux listes.

Une « liste en compréhension » permet ici d'avoir rapidement une liste Z formée des logarithmes des points de X.

Une commande de tracé : la liste des points d'abscisses dans X, ordonnées dans Z, largeur fine.