Université de Bordeaux — **Algèbre et calcul formel – Agrégation**
Mathématiques — 2022–2023

FEUILLE D'EXERCICES n° 9

Représentations graphiques

Exercice 1 – [Commandes de base]
Dans l'aide (`plot?`), on trouve des explications et des exemples d'utilisation des commandes graphiques `plot`, `line`, `point`, `text`, etc. Voyons ici quelques exemples.

1) Exécuter les commandes suivantes.

```
plot(sin)
plot(sin,(0,10))
plot(sin(1/x))
```

La fonction `plot` semble s'appliquer à l'intervalle $[-1, 1]$ par défaut. On peut aussi bien utiliser `sin` ou `sin(x)` dans `plot`.

2) Pour dessiner un nuage de points :

```
point([(-1,-1),(7,0),(1,5.3),(6,8)])
```

ou bien, si l'on n'y voit pas très bien :

```
point([(-1,-1),(7,0),(1,5.3),(6,8)],color='red',pointsize=20)
```

On peut aussi donner un nom à cette figure

```
p = point([(-1,-1),(7,0),(1,5.3),(6,8)],color='red',pointsize=20)
```

3) Pour dessiner des segments, on dispose de la commande `line`.

```
l = line([(-1,-1),(7,0),(1,5.3),(6,8)]); l
```

On peut alors dessiner `p` et `l` sur la même figure.

```
lp = p+l; lp
```

On peut aussi ajouter du texte.

```
t  = text('A',(-1.2,-1),color='red')
t += text('B',(7,0.35),color='red')
t += text('C',(0.8,5.3),color='red')
t += text('D',(6.2,8),color='red')
lp + t
```

4) Pour un exemple plus joli, exécuter les commandes suivantes (prises dans l'aide).

```
p = plot(x^2, (-0.5, 1.4)) + line([(0,0), (1,1)], color='green')
p += line([(0.5,0.5), (0.5,0.5^2)], color='purple')
p += point(((0,0), (0.5,0.5), (0.5,0.5^2), (1,1)), color='red', pointsize=20)
p += text('A', (-0.05, 0.1), color='red')
p += text('B', (1.01, 1.1),  color='red')
p += text('C', (0.48, 0.57), color='red')
p += text('D', (0.53, 0.18), color='red')
p
p.show(axes=False, xmin=-0.5, xmax=1.4, ymin=0, ymax=2)
```

5) Il y a différentes manières d'indiquer les bornes d'un graphe. Dans l'aide, on trouve (entre autre) les commandes suivantes.

```
plot(x^2, x, -1,2)
plot(x^2,-1,2)
plot(x^2, xmin=-1,xmax=2)
```

6) Si on veut tracer des fonctions un peu plus compliquées, on peut les définir séparément, puis les appeler dans `plot`. Par exemple

```
f(x) = sin(x+3) - .1*x^3
plot(f,-2,2)
```

On peut aussi utiliser `plot(f(x),x,-2,2)` ou encore `plot(f(x),-2,2)`. Attention cependant, ça ne marche que parce que `x` est de type `var` par défaut en sage. Si vous re-définissez `x` à un moment dans votre code ou que vous voulez utiliser une autre variable, il faudra la déclarer `var` avant. Essayez par exemple

```
plot(f(y),y,-1,2)
y = var('y')
plot(f(y),y,-1,2)

x = 3
plot(f(x),x,-1,2)
```

(La dernière commande affiche bien quelque chose, mais ce n'est pas ce qu'on veut : elle affiche juste $f(3)$ sur tout l'intervalle.)

7) On peut aussi dessiner deux courbes (ou plus) sur un même graphe, en mettant les fonctions dans une liste.

```
plot([sin(x), cos(2*x)*sin(4*x)], -pi, pi)
```

Exercice 2 – [Mesures expérimentales de temps de calcul]

Dans cet exercice, on veut déterminer empiriquement la complexité asymptotique (en nombre d'opérations binaires) de l'algorithme de factorisation de sage. Pour ce faire, on va mesurer empiriquement le temps de cet algorithme pour différentes tailles d'entrées, et on va partir du principe que le temps effectif est proportionnel à la complexité binaire des algorithmes.

Comme la complexité de l'algorithme de factorisation dépend probablement beaucoup des facteurs premiers de l'entrée et pas simplement de sa taille, on ne va mesurer la complexité que sur des entrées obtenues en multipliant deux grands nombres premiers. Cela va nous donner une estimation de la complexité de l'algorithme dans le pire cas. En pratique, l'algorithme pourrait être significativement plus rapide sur des entrées choisies au hasard.

1) Écrire une fonction qui prend en entrée un entier $N > 0$, génère deux entiers premiers au hasard de taille $N//2$ (i.e., avec $N//2$ bits), puis les multiplie pour obtenir un entier x et enfin mesure le temps de l'appel à la fonction `factor` de Sage sur x, et renvoie ce temps.
Rappels du TD07 : Pour mesurer le temps, on pourra utiliser la commande `import time` (à utiliser une seule fois), puis la commande `t = time.time()`, qui donne le temps à un moment de l'execution. En faisant

```
t1 = time.time()
bla
blo
t2 = time.time()
print(t2-t1)
```

on affiche le temps qu'il s'est écoulé entre le début et la fin des commandes `blablo`.

2) Écrire une fonction qui prend en entrée deux entiers N et k, et répète k fois la fonction précédente sur l'entier N, puis renvoie la moyenne de tous les temps observés sur les k répétitions.

3) Essayez de déterminer une bonne valeur de k pour votre fonction précédente : on veut k pas trop gros pour que la fonction ne soit pas trop longue, mais suffisamment grand pour que le résultat en sortie ne varie pas trop d'une exécution à l'autre.

4) De même, déterminez quelle valeur maximal de N vous pouvez atteindre dans votre fonction.

5) Une fois la valeur maximale $N_{\max}$ de N et la valeur optimale k_{opt} de k déterminées, créez une liste de la forme $[(N_i, t_i)]$ de taille environ 10 à 20 points, où N_i varie entre 2 et $N_{\max}$, et t_i et la sortie de votre fonction de la question 2 sur l'entrée N_i et k_{opt}.

6) Tracez votre liste de points, avec t_i en fonction de N_i. Pouvez-vous dire si la complexité est polynomiale, exponentielle, sous-exponentielle, ... ?

En pratique, c'est difficile de différencier une courbe d'une autre courbe, à part la droite, qui se distingue bien des autres. C'est pourquoi on va changer d'échelle et prendre des log (voire des log log), pour essayer d'obtenir une droite.

7) Quelle complexité vous attendez-vous à obtenir : polynomial ($t \approx N^c$ pour une constante c), exponentielle ($t \approx \exp(cN)$ pour une constante c), ou sous-exponentiel ($t \approx \exp(cN^d)$ pour deux constantes c et d, avec $0 < d < 1$) ?

8) Selon votre réponse à la question précédente, que devez vous tracer pour obtenir une droite ? (t, ou $\log t$ ou $\log\log t$, en fonction de N, ou $\log N$, ou $\log\log N$) Tracez et vérifiez que vous obtenez bien une droite.

9) Si vous avez réussi à obtenir une droite à la question précédente, utilisez-la pour estimer le paramètre c (ou d dans le cas sous-exponentiel) de votre complexité.

Exercice 3 – [Courbes paramétrées]

1) Comment dessiner une courbe paramétrée ? Avec `parametric_plot`. Voici donc une façon de dessiner un cercle :

```
parametric_plot( [cos(x), sin(x)], (x, 0, 2*pi))
```

Un exemple plus amusant :

```
parametric_plot([cos(x) + 2*cos(x/4), sin(x) - 2*sin(x/4)], (x,0, 8*pi))
```

Un exemple où l'on recolle deux branches :

```
P1 = parametric_plot((3*x/(x^3+1), 3*x^2/(x^3+1)), (x,-0.8,10))
P2 = parametric_plot((3*x/(x^3+1), 3*x^2/(x^3+1)), (x,-10,-1.2))
P1 + P2
```

2) Soit $\mathcal{C}$ la courbe d'équation paramétrée

$$\begin{cases} x = 4t^3 \\ y = 3t^4 \end{cases}$$

Dessiner la courbe $\mathcal{C}$.

3) Restent encore les courbes paramétrées en coordonnées polaires : `polar_plot`.

```
polar_plot(sin(5*x)^2, (x, 0,2*pi), color='red')
```

Dessiner la courbe d'équation polaire

$$r = \frac{\cos(2\theta)}{\cos\theta}$$

4) Une autre commande intéressante : `implicit_plot`. Par exemple, essayer

```
var("x y")
implicit_plot(x^2+y^2-2, (x,-3,3), (y,-3,3))
```

On peut également définir la fonction préalablement :

```
f(x,y) = x^2 + y^2 - 2
implicit_plot(f(x,y), (x,-3,3), (y,-3,3),fill=true)
```

Exemple d'une figure composée de plusieurs graphes (issue de l'aide)

```
G = Graphics()
counter = 0
for col in colors.keys():
     G += implicit_plot(x^2+y^2==1+counter*.1, (x,-4,4),(y,-4,4),color=col)
     counter += 1
G.show(frame=False)
```

Ici, `Graphics` désigne le graphe vide. Il sert à donner une première valeur à `G`, avant de le faire varier dans la boucle.

5) Dessiner sur un graphe un triangle et son cercle circonscrit.

6) Dessiner la courbe d'équation

$$x^3 + y^3 - 3xy.$$

Exercice 4 – [Mesures expérimentales : le retour]

Cette fois, on va vérifier expérimentalement la complexité d'un algorithme que vous avez implémenté et dont vous connaissez la complexité asymptotique a priori : l'algorithme de Strassen du TD6. Pour rappel, c'est un algorithme récursif de multiplication matricielle, qui permet d'atteindre une complexité inférieure au $O(n^3)$ de la multiplication matricielle naïve (c'est un peu l'analogue de Karatsuba, pour la multiplication matricielle). L'algorithme est le suivant :

Soient A et B deux matrices $n \times n$ sur un corps K, où n est une puissance de 2. Nous voulons calculer le produit $C = AB$. On subdivise A, B et C en quatre blocs de taille $n/2 \times n/2$:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} \quad \text{et} \quad C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}.$$

On calcule ensuite

$$\begin{cases} P_1 &= A_1(B_2 - B_4) \\ P_2 &= (A_1 + A_2)B_4 \\ P_3 &= (A_3 + A_4)B_1 \\ P_4 &= A_4(B_3 - B_1) \\ P_5 &= (A_1 + A_4)(B_1 + B_4) \\ P_6 &= (A_2 - A_4)(B_3 + B_4) \\ P_7 &= (A_1 - A_3)(B_1 + B_2) \end{cases}$$

puis on remarque que

$$\begin{cases} C_1 &= P_5 + P_6 - P_2 + P_4 \\ C_2 &= P_1 + P_2 \\ C_3 &= P_3 + P_4 \\ C_4 &= P_5 - P_7 + P_1 - P_3 \end{cases}$$

ce qui nous permet de calculer la matrice $C = AB$.

1) Si vous ne l'aviez pas déjà fait, implémentez l'algorithme récursif obtenu en utilisant les observations ci-dessus : c'est l'algorithme de Strassen. (Si vous l'aviez déjà fait, récupérez votre fonction du TD06)

2) Quelle est la complexité (en nombre d'opérations dans le corps K) de cet algorithme, en fonction de la dimension n de la matrice ?

3) Vérifiez cette complexité expérimentalement. Remarque : ce qu'on mesure expérimentalement, c'est le temps de l'algorithme, pas le nombre d'opérations dans K. Pour que nos mesures ne soient pas trop faussées par le coût des opérations dans K, on pourra choisir par exemple $K = \mathbb{F}_5$, pour que les opérations dans K aient toutes un coût constant en terme d'opérations binaires.