TP4 : Les procédures sous Maple12 et la représentation graphique

Cette séance de TP poursuit la familiarisation avec Maple12. Le chapitre 2 du cours (méthodes approchées pour le calcul des zéros d'équations non linéaires) fournira le support théorique à ce TP. Ouvrez Maple12 pour commencer. L'initiation à la rédaction de *procédures*, en particulier de *procédures récursives*, constituera la trame de cette séance. Le tracé de courbes fractales (flocon de Von Koch, courbe de Lévy) mettra en œuvre les techniques acquises ici.

Avant de commencer, voici quelles sont les principales structures de données sous Maple12. Contrairement à l'environnement MATLAB, nous disposons ici d'un champ de structures de données plus vaste que celui des structures à base de tableaux, ceci s'avérant nécessaire en vue de l'objectif assigné au logiciel, à savoir le calcul symbolique.

- Les *suites* (finies bien sûr) se déclarent ainsi sous Maple12 :
  `> S := 5,6,7,n,8,n,9,n+1;`
  On accède au terme d'indice k dans la suite (k=1,...,N si celle ci est de longueur N) avec l'instruction
  `> S[k];`
  (faites par exemple le test avec la suite déclarée ci-dessus, puis refaites le une fois spécifiée une valeur de la variable non déclarée n). La suite vide se déclare comme `S:= Null;`.
- Les *listes* (ordonnées) sont assimilables aux tableaux à une ligne (et N colonnes) tels qu'ils sont déclarés sous MATLAB (excepté le fait que l'on doive ici séparer les entrées successives par des virgules) :
  `> L := [5,6,7,n,8,n,9,n+1];`
  `> L[k];`
  (faites encore le test). La liste vide se déclare comme `L:=[ ];`.
- Les *ensembles* correspondent à des listes, mais cette fois non ordonnées. Un ensemble (fini) E se déclare *via* :
  `> E := {5,6,7,n,8,n,9,n+1};`
  L'ensemble vide emptyset se déclare comme `emptyset :={ };`.

Familiarisez vous avec l'instruction op qui renvoie les opérandes d'une expression e (lorsque l'on précise leur position), ou le nombre d'opérantes (nops), voire l'expression e simplifiée (op(e)), par exemple :

```
> op(0,5,6,7,n,8,n,9,n+1);
> op(0,[5,6,7,n,8,n,9,n+1]);
> op(0,{5,6,7,n,8,n,9,n+1});
> op(4,5,6,7,n,8,n,9,n+1);
> op(7,[5,6,7,n,8,n,9,n+1]);
```

```
> op(8,{5,6,7,n,8,n,9,n+1});
> op(9,{5,6,7,[n,8,n,9,n+1]});
> op(3..4,{5,6,7,[n,8,n,9,n+1]});
> op(4..6,{5,6,7,n,8,n,9,n+1});
> op(4..6,{5,6,7,[n,8,n,9,n+1]});
> op([5,6,7,n,{8,n,9,n+1}]);
> nops([5,6,7,n,{8,n,9,n+1}]);
```

Pour appliquer une fonction (disons `g`, préalablement déclarée) à une suite, une liste, ou un ensemble (et donc calculer la suite, la liste ou l'ensemble constituées des images par `g` des éléments de la suite, de la liste ou bien de l'ensemble initial), on utilise l'instruction `map`. Exemple :

```
> g:= x-> x^3 - sin(x);
> map(g,S);
> map(g,L);
> map(g,E);
```

Recommencez après avoir assigné une valeur numérique à `n`. Entrainez vous.

Dans ce TP, vous allez apprendre à rédiger des programmes autonomes. Supposons que l'on souhaite rédiger une `procedure` sous `Maple12`, c'est-à-dire créer une fonction `Maple` qui, étant donnée un certain nombres d'`arguments` (ou encore « données » en `input`, par exemple `argument1,argument2,...,argumentk`), renvoie, une fois la procédure lancée, une sortie (`NomProcedure`) correspondant au dernier résultat fourni par l'exécution du code sous les données initiales que sont `argument1`,...,`argumentk`. Le schéma de la rédaction d'une telle procédure dans une zone de code est donc le suivant :

```
NomProcedure := proc(argument1,...,argumentk)
instruction1;
instruction2;
...
instructionk;
end proc;
```

Exercice 1 (Génération de suites récurrentes à un ou plusieurs pas). Le but de cet exercice est de réaliser des procédures récursives (un seul `argument` entier noté `n`, plus éventuellement des arguments supplémentaires figurant des valeurs de paramètres dont dépend la suite à générer) permettant de générer une suite régie par une récurrence à un ou plusieurs pas. Le terme *récursive* signifie que la procédure s'auto-appelle lors de son exécution. Pour éviter que l'exécution du code ne boucle indéfiniment, il faut bien sûr être soigneux de déclarer les « cas initiaux » !

(1) Vérifiez que la procédure suivante permet de générer une suite arithmétique de raison `4` et de premier terme `3` :

```
fonction:=proc(n)
if n=0 then
3;
else
fonction(n-1) + 4;
end if;
end proc;
```

Pourquoi est-il essentiel de faire figurer la boucle `if n=0 then else ... end if;` dans le synopsis ? Déduisez de cela la construction d'une procédure récursive

`fonction1TP4 := proc(raison,init,n)`

qui, étant données les trois arguments `raison`, `init`, `n`, génère la suite arithmétique de premier terme `init` et de raison `raison`.

(2) Sur le modèle établi à la question **1**, rédigez une procédure récursive

`FibonacciRec:=proc(n)`

permettant de calculer le `n`-ième nombre de Fibonacci. On rappelle que la suite des nombres de Fibonacci est la suite $(F_n)_{n\geq 0}$ telle que $F_0 = F_1 = 1$ et

$$F_n = F_{n-2} + F_{n-1} \qquad \forall\ n \geq 2.$$

Adaptez ce code pour l'insérer dans une procédure récursive

`fonction2TP4 :=proc(a,b,init0,init1,n)`

qui, étant donnés les cinq arguments `a,b,init1,init2,n`, génère la suite $(u_n)_{n\geq 0}$ initiée à $u_0 =$ `init0` et $u_1 =$ `init1` et régie ensuite par la récurrence linéaire à deux pas :

$$u_n = a\,u_{n-1} + b\,u_{n-2} \qquad \forall\, n \geq 2.$$

(3) En utilisant la commande `time` (combinée avec la commande `seq` permettant de générer des suites ou des listes), générez la liste `L` des temps de calcul CPU pris par l'exécution de `FibonacciRec (i)`, `i=20..35`. En utilisant `listplot` (sous `with (plots);`), affichez (en fonction de l'indice `i`) la liste obtenue en transformant `L` par l'application `log`. Que constatez vous ?

(4) Pour chaque entier n supérieur ou égal à 2, soit c_n le nombre d'appels au programme `FibonacciRec` impliqués dans `FibonacciRec(n)`. Montrez que la suite $(c_n)_{n\geq 2}$ vérifie la relation $c_n = c_{n-1} + c_{n-2} + 1$, puis déduisez en que $c_n = 2F_n - 1$. Lorsque n tend vers $+\infty$, prouvez en utilisant cette fois la relation matricielle

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

et votre cours d'Algèbre 3 que $F_n \sim ((1+\sqrt{5})/2)^n$. Comparez la pente visible sur le graphe obtenu précédemment avec le nombre d'or $(1+\sqrt{5})/2$. Pouviez vous prévoir le résultat observé ainsi ?

(5) Modifiez la syntaxe de la procédure `FibonacciRec` en utilisant l'instruction additionnelle `option remember`, sous la forme

`FibonacciRec := proc(n) option remember;`

Testez là maintenant avec des valeurs de `n` supérieures à `35`. Que constatez vous ? Modifiez la procédure obtenue en une procédure `FibonacciRecbis` de manière à ce que se trouvent affichées en sortie toutes les valeurs intermédiaires $F_0, ..., F_n$ (au lieu de simplement la valeur finale F_n). Testez là pour $n = 30$ et comparez avec les temps de calculs impliqés dans l'utilisation de `FibonacciRec` aux questions **3** et **4**.

Sauvez votre feuille de travail comme `TP4exo1` dans votre répertoire `TPMaple12` puis fermer cette feuille de travail.

EXERCICE 2 (approximation de zéros par des méthodes itératives). Ouvrez une nouvelle feuille de travail.

(1) Sur le modèle de ce qui a été fait à l'exercice 1, construisez une procédure récursive [1]

`fonction3exactTP4 := proc(a,b,c,init0,init1,n)`

qui, étant donnés 6 arguments `a,b,c,init0,init1,n` (cinq rationnels, un entier naturel)

- soit calcule exactement, lorsque cela est possible (c'est-à-dire si l'exécution du code ne fait jamais apparaitre de division par 0), le terme d'ordre `n` dans la suite initiée avec $u_0 =$ `init0`, $u_1 =$ `init1`, et régie par la relation inductive :

$$u_n = a + \frac{b}{u_{n-1}} + \frac{c}{u_{n-1}u_{n-2}} \; ; \qquad (\dagger)$$

- soit retourne le message d'erreur
`Error, division by zero`
dès que surgit dans l'exécution du code une division par 0. Testez cette procédure en déclarant les arguments `a:=108, b:=-815, c:=1500, init0:=11/3, init1:=43/11, n:=30`.

Modifiez la en une procédure `fonction3exactTP4bis` qui affiche en sortie toutes les valeurs u_k intermédiaires.

(2) Modifiez la procédure `fonction3exactTP4` pour construire une procédure `fonction3approcheeTP4` qui cette fois, toujours avec les mêmes arguments, calcule (toujours de manière récursive) de manière approchée les u_n de proche en proche. Modifiez la en une procédure `fonction3exactTP4bis` qui affiche en sortie toutes les valeurs u_k intermédiaires. Testez ces procédures `fonction3approcheeTP4` et `fonction3approcheeTP4bis` avec les mêmes arguments déclarés qu'à la question **1** et comparez avec ce que donnent les procédures `fonction3exactTP4` et `fonction3exactTP4bis` réalisées à la question **2**. Que constatez vous de manière évidente ?

(3) En utilisant l'instruction `solve`, trouvez avec `Maple12` les trois racines (exactes) du polynôme $X^3 - 108X^2 + 815X - 1500$.

(4) On fixe maintenant, comme dans les questions **1** et **2**, `a:=108, b:=-815, c:=1500, init0:=11/3, init1:=43/11`. On pose $a_0 = 1$ et, pour tout $n \in \mathbb{N}$, $a_{n+1} = u_0 \cdots u_n$. En utilisant la relation inductive (†), vérifiez (mathématiquement) que la suite $(a_n)_{n\geq 0}$ obéit à la relation récurrente (à trois pas) :

$$a_{n+3} - 108\, a_{n+2} + 815\, a_{n+1} - 1500\, a_n = 0 \quad \forall\, n \in \mathbb{N}. \qquad (\dagger\dagger)$$

Vérifiez que les suites $(3^n)_{n\geq 0}$, $(5^n)_{n\geq 0}$ et $(100^n)_{n\geq 0}$ vérifient aussi la relation récurrente (††). Chargez le package `LinearAlgebra` et consultez

1. Pensez à y intégrer l'instruction `option remember` comme à la question **5** de l'exercice 1.

l'aide de `LinearSolve` pour trouver trois nombres rationnels α, β, γ de manière à ce que :

$$\begin{aligned} a_0 &= \alpha + \beta + \gamma \\ a_1 &= 3\,\alpha + 5\,\beta + 100\,\gamma \\ a_2 &= 9\,\alpha + 25\,\beta + 10000\,\gamma. \end{aligned}$$

Déduisez en

$$\forall\, n \in \mathbb{N},\ a_n = \frac{2}{3}\,3^n + \frac{1}{3}\,5^n.$$

Définissez (sous `Maple12`) une fonction `f` qui à n associe a_n. Comment s'exprime la fonction $n \mapsto u_n$ en fonction de `f` ? Déclarez cette nouvelle fonction $n \mapsto u_n$ sous `Maple12`. En utilisant l'instruction `simplify`, vérifiez avec `Maple12` que la suite $(u_n)_{n\geq 0}$ est bien une suite croissante, majorée par 5. Ceci vous permet-il de justifier ce que vous aviez observé à la question **1** concernant le comportement asymptotique de la suite $(u_n)_{n\geq 0}$?

Sauvez votre travail comme le fichier `TP4exo2` dans votre répertoire `TPMaple12`, puis fermez cette feuille de travail.

EXERCICE 3 (extraction d'une racine carrée par l'algorithme « de Babylone »). L'algorithme dit « de Babylone » constitue depuis l'Antiquité l'illustration la plus célèbre de la méthode de Newton. Cet exercice est centré autour de cette méthode classique. Ouvrez une nouvelle feuille de travail.

(1) Montrez que la suite itérative conduisant à l'approximation de $\sqrt{2}$ par la méthode de Newton est régie par la relation inductive :

$$x_n = \frac{x_{n-1}}{2} + \frac{1}{x_{n-1}} \quad \forall\, n \in \mathbb{N}^*.$$

Vérifiez que si cette suite $(x_n)_{n\geq 0}$ est initiée à $x_0 = 7/5$, c'est une suite de rationnels et que l'on a

$$|x_{n+1} - \sqrt{2}| \leq |\sqrt{2} - x_n|^2 \quad \forall\, n \in \mathbb{N}. \qquad (*)$$

Quel terme de cette suite $(x_n)_{n\geq 0}$ convient-il alors de calculer pour déterminer $\sqrt{2}$ avec 1000 décimales exactes ?

(2) Rédigez deux procédures pour générer la suite de Babylone $(x_n)_{n\geq 0}$ (générée avec $x_0 =$ `init` et régie par la relation inductive $(*)$:
   - l'une récursive `procNewtonRec:=proc(init,n)`
   - l'autre directe `procNewton:=proc(init,n)`

   retournant toutes deux le terme d'ordre `n` de la suite $(x_n)_{n\geq 0}$ lorsque celle ci est initiée au nombre `init` (que l'on déclarera comme un nombre flottant). Testez ces deux procédures avec `init`=1.4 et `n`=10 en prenant `Digits:=1001`. Comparez le résultat avec `evalf(sqrt(2))`.

(3) Transformez la procédure `procNewton` en une autre, `procNewton2`, qui commence la calcul avec une précision aussi faible que possible (`Digits:=2`) et double la précision à chaque itération. Quelle valeur de `n` faut-il cette fois prendre pour retrouver $\sqrt{2}$ avec 1000 décimales exactes ? Comparez les temps d'exécution de la procédure `procNewton` (avec `init`=1.4 et `n`=10 sous `Digits:=1001`) et de la nouvelle procédure `procNewton2`, avec toujours `init`=1.4, mais cette fois `n`=12.

Sauvez votre travail cette fois comme le fichier `TP4exo3` (toujours dans le répertoire `TPMaple12`) et fermez ensuite cette feuille de travail.

EXERCICE 4 (une approximation de π par des suites adjacentes). Ouvrez une nouvelle feuille de travail.

(1) Réalisez une procédure `approxPi1TP4:=proc(n)` (basée sur une boucle `DO`) générant simultanément les deux suites $(u_n)_{n\geq 0}$ et $(v_n)_{n\geq 0}$ de nombres strictement positifs initiées respectivement par $u_0 = 1.$, $v_0 = 2.$ (attention de déclarer ces entrées comme des nombres flottants !) et obéissant au couple de relations inductives :

$$\begin{aligned} u_{n+1} &= \frac{u_n + v_n}{2} \\ v_{n+1} &= \sqrt{u_{n+1} v_n}. \end{aligned}$$

Lancez cette procédure sous par exemple `Digits:=20` avec `n=50`. Que constatez vous ? On admet que les deux suites $(u_n)_{n\geq 0}$ et $(v_n)_{n\geq 0}$ sont adjacentes et ont pour limite commune le nombre $\sqrt{27}/\pi$. Modifiez votre procédure en une procédure `approxPi2TP4:=proc(n)` qui fournisse l'approximation de π déduite de l'égalité approchée $(u_n + v_n)/2 \simeq \sqrt{27}/\pi$.

(2) Modifiez la procédure `approxPi2TP4` en une procédure

`approxPi3TP4:=proc(epsilon,n)`

qui, étant donné un seuil `epsilon` fixé, retourne l'approximation de π déduite de l'égalité approchée $(u_n + v_n)/2 \simeq \sqrt{27}/\pi$ dès que la condition

$$\frac{|u_n - v_n|}{u_n + v_n} \leq \epsilon$$

est satisfaite (et affiche le nombre `i`$\leq$`n` d'itérations de la boucle `DO` nécessaires).

(3) Réalisez une procédure inductive `approxPiRecTP4:=proc(n)` générant simultanément les deux suites $(u_n)_{n\geq 0}$ et $(v_n)_{n\geq 0}$ de la question **1**, ce de manière exacte cette fois (on prend $u_0 = 1$ et $v_0 = 2$, déclarés cette fois comme rationnels et non plus comme flottants). Comparez, pour `n`=12 (attention surtout à ne pas prendre de valeur plus grande, car le temps de calcul risque d'exploser !) le temps d'exécution de cette nouvelle procédure `approxPiRecTP4(12)` avec celui de la procédure `approxPi1TP4(12)` (dans laquelle u_0 et u_1 sont déclarés comme les entiers 1 et 2 et non plus comme les flottants 1.0 et 2.0). Que constatez vous ?

Sauvez votre feuille de travail comme `TP4exo4` (dans `TPMaple12`) et fermez cette feuille.

EXERCICE 5 (un modèle de courbe fractale : le flocon de Von Koch). Les courbes planes qui ont la propriété d'*autosimilarité*, c'est-à-dire qui, regardées à la loupe avec n'importe quel grossissement, reproduisent à toutes les échelles le même motif, se prêtent (en ce qui concerne leur tracé) à la réalisation de procédures récursives. Le « flocon de Von Koch », introduit vers 1900 par le mathématicien suédois Helge von Koch (1870-1924), en est une illustration ; regardez par curiosité avant de commencer le site dédié :

`http://fr.wikipedia.org/wiki/Flocon_de_Koch`

(1) Soient a et b deux nombres complexes, définissant un segment (noté $[a,b]$) du plan complexe. On introduit les trois nombres complexes :

$$\begin{aligned} u(a,b) &:= a + \frac{b-a}{3} = \frac{2a+b}{3} \\ v(a,b) &:= b - \frac{b-a}{3} = \frac{2b+a}{3} \\ w(a,b) &:= u(a,b) + e^{i\pi/3}(v(a,b) - u(a,b)) \end{aligned}$$

et l'on note Φ l'application qui au segment $[a,b]$ (d'origine a et d'extrémité b) associe la ligne brisée $\Phi([a,b])$ joignant (dans cet ordre) les quatre points $a, u(a,b), w(a,b), v(a,b), b$. Si $\mathcal{L}$ est une ligne brisée de sommets d'affixes $a_1, a_2, ..., a_N$, la ligne brisée $\Phi(\mathcal{L})$ désigne par extension la ligne brisée obtenue en mettant bout à bout les diverses lignes brisées $\Phi([a_j, a_{j+1}])$ pour $j = 1, ..., N-1$. Exprimez en fonction de N (nombre de sommets de la ligne brisée $\mathcal{L}$) le nombre de sommets de la ligne brisée $\Phi(\mathcal{L})$. Réalisez une procédure

`VonKoch1TP4:=proc(L)`

qui, étant donnée une liste `L` de nombres complexes, considérée comme la liste `[a1,...,aN]` des affixes des sommets successifs d'une ligne brisée $\mathcal{L}$, renvoie (dans l'ordre, et de manière approchée) la liste des affixes des sommets de la ligne brisée $\Phi(\mathcal{L})$.

(2) En utilisant la procédure `VonKoch1TP4`, réalisez une nouvelle procédure

`VonKoch2TP4:=proc(n)`

qui renvoie (dans l'ordre) la liste des affixes des sommets de la ligne brisée $\Phi^{[n]}([0,1])$, où $\Phi^{[n]}$ désigne l'application Φ itérée `n` fois. Vu que le nombre de sommets de la ligne brisée $\Phi^{[n]}([0,1])$ croit vers l'infini comme $2 \times 4^{\mathtt{n}}$ (dites pourquoi), vous ne testerez cette procédure `VonKoch2TP4` que sur des valeurs de `n` entre 0 et 5. En utilisant les instructions

```
> with (plots);
> L:=VonKoch2TP4(n):
> complexplot(L,style = line, scaling = constrained);
```

affichez la ligne brisée obtenue en itérant `n` fois Φ à partir du segment $[0,1]$. Veillez à ne prendre comme valeurs de `n` que des valeurs entre 1 et 7 (notez que $2 \times 4^7 = 32768$ est déjà très grand !). Que se passe-t-il si vous remplacez la dernière instruction par :

```
> complexplot(L,style=line);
```

La courbe ainsi obtenue est une approximation du flocon de Von Koch ; ce flocon de Von Koch est un exemple de courbe fractale, continue mais ne présentant de tangente en aucun point [2] ; c'est en fait la limite uniforme des courbes ainsi tracées – en fonction de `n` – lorsque `n` tend vers l'infini).

2. Vous verrez plus tard dans votre cursus mathématique que pareille courbe fractale n'est pas ≪ rectifiable ≫, et que l'on ne peut donc pas parler de ≪ longueur ≫ du flocon de Von Koch. La nature fourmille de tels modèles fractaux (en relation avec ce que l'on appelle le ≪ chaos dynamique ≫) : pensez par exemple à la découpe d'une côte volcanique telle celle du Groendland, à celle de la ligne d'arête des aiguilles de Chamonix ... Vous trouverez aussi beaucoup de modèles (mathématiques cette fois) d'images fractales sur le `web` (courbes de Mandelbrojt, frontières de domaines de Julia, *etc.*).

(3) Réalisez une procédure cette fois récursive

`VonKoch3TP4 :=proc(a,b,n)`

retournant la liste (dans l'ordre) des affixes des sommets de la ligne brisée $\Phi^{[n]}([a,b])$. En prenant $a = 0$ et $b = 1$, retrouvez (en visualisant les résultats avec `complexplot` plutôt qu'en les faisant afficher numériquement !) le résultat `L` de l'instruction

```
>L:= VonKoch2TP4(n):
```

Comparez les temps CPU nécessaires par les deux instructions

```
> L:= VonKoch3TP4 (0,1,n);
> L:= VonKoch2TP4 (n);
```

pour les valeurs de `n` entre 3 et 8 (si toutefois vous parvenez jusqu'à `n` = 8 avec la seconde instruction [3]).

Sauvez votre feuille de travail comme `TP4exo5` dans votre répertoire `TPMaple12`, puis fermez cette feuille.

EXERCICE 6 (un autre modèle de courbe fractale : la courbe de Lévy). Si elle porte le nom du probabiliste français Paul Lévy, la *courbe de Lévy* (qui est un autre exemple célèbre de courbe fractale) a été introduite par les mathématiciens Ernesto Cesàro (italien) en 1906 et Georg Faber (allemand) en 1910. Regardez par curiosité le site dédié sur `wikipedia`. Ouvrez une nouvelle feuille de travail sous `Maple12`.

(1) Soient a et b deux nombres complexes, définissant un segment du plan complexe. On introduit le nombre complexe :

$$r(a,b) = \frac{a+b}{2} + i\frac{b-a}{2}.$$

et l'on note Ψ l'application qui à ce segment (d'origine a et d'extrémité b) associe la ligne brisée $\Psi([a,b])$ joignant (dans cet ordre) les trois points $a, r(a,b), b$. Si $\mathcal{L}$ est une ligne brisée de sommets d'affixes $a_1, a_2, ..., a_N$, la ligne brisée $\Psi(\mathcal{L})$ désigne par extension la ligne brisée obtenue en mettant bout à bout les diverses lignes brisées $\Psi([a_j, a_{j+1}])$ pour $j = 1, ..., N-1$. Exprimez en fonction de N (nombre de sommets de la ligne brisée $\mathcal{L}$) le nombre de sommets de la ligne brisée $\Psi(\mathcal{L})$. Comme dans la question **1** de l'exercice 5, réalisez une procédure

`Levy1TP4 := proc(L)`

qui, étant donnée une liste `L` de nombres complexes, considérée comme la liste `[a1,...,aN]` des affixes des sommets successifs d'une ligne brisée $\mathcal{L}$, renvoie (dans l'ordre, et de manière approchée) la liste des affixes des sommets de la ligne brisée $\Psi(\mathcal{L})$.

(2) Réalisez une procédure récursive

`Levy2TP4 := proc(epsilon,L)`

qui, étant donnée une liste d'affixes `L` (figurant la liste des sommets ordonnés d'une ligne brisée $\mathcal{L}$) renvoie la liste (ordonnée) des affixes des sommets de la ligne brisée $\Psi^{[n]}(\mathcal{L})$ dès l'instant où le pas de cette ligne brisée devient strictement inférieur au seuil `epsilon`.

3. Notéz que $2 \times 4^8 = 131072$, ce qui est vraiment très grand pour la taille d'une liste de flottants sous `Maple12` !

(3) En utilisant `complexplot` (après avoir déclaré l'environnement `plots` par l'instruction `with(plots);`), représentez la liste `L` obtenue *via*

```
> L:=Levy2TP4(.03,[I,1,I]):
```

Sauvez votre feuille de travail comme `TP4exo6` dans votre répertoire `TPMaple12`, puis fermez cette feuille.