Le paquet *luadraw* 1.0

Dessin 2d avec lua (et tikz)

P. Fradin

11 mars 2025

Résumé

Le paquet *luadraw* définit l'environnement du même nom, celui-ci permet de créer des graphiques mathématiques en utilisant le langage Lua. Ces graphiques sont dessinés au final par tikz (et automatiquement sauvegardés), alors pourquoi les faire en Lua? Parce que celui-ci apporte toute la puissance d'un langage de programmation simple, efficace, capable de faire des calculs, tout en utilisant les possibilités graphiques de tikz.

Table des matières

Table des figures

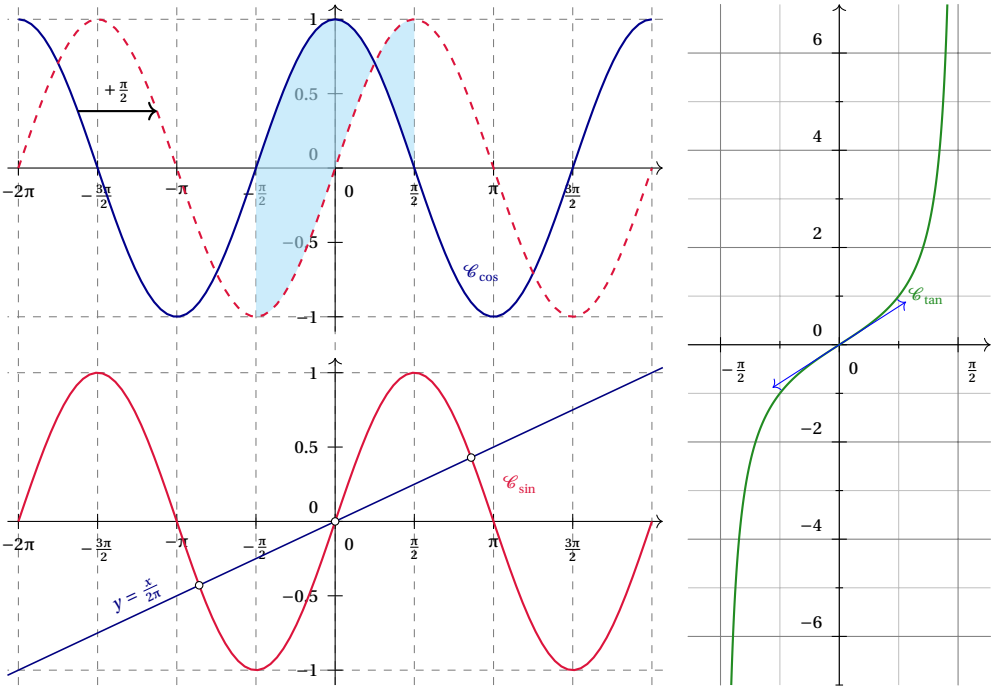

FIGURE 1 – Un premier exemple : trois sous-figures dans un même graphique

I Introduction

1) Prérequis

- Dans le préambule, il faut déclarer le package *luadraw* : `\usepackage[option globale]{luadraw}`
- La compilation se fait avec LuaLatex **exclusivement**.
- Les couleurs dans l'environnement *luadraw* sont des chaînes de caractères qui doivent correspondre à des couleurs connues de tikz. Il est fortement conseillé d'utiliser le package *xcolor* avec l'option *svgnames*.

Option globale du paquet : *noexec*.

Lorsque cette option globale est mentionnée alors la valeur par défaut de l'option *exec* pour l'environnement *luadraw* sera false (et non plus true).

Ce paquet charge le module *luadraw_graph2d.lua*, et fournit l'environnement *luadraw* qui permet de faire des graphiques en Lua.

Lorsqu'un graphique est terminé il est exporté au format tikz, donc ce paquet charge également le paquet *tikz* ainsi que les librairies :

- *patterns*
- *plotmarks*
- *arrows.meta*
- *decorations.markings*

Les graphiques sont créés dans un environnement *luadraw*, celui-ci appelle *luacode*, c'est donc du **langage Lua** qu'il faut utiliser dans cet environnement :

```
\begin{luadraw}{ name=<filename>, exec=true/false, auto=true/false }
-- création d'un nouveau graphique en lui donnant un nom local
local g = graph:new{ window={x1,x2,y1,y2,xscale,yscale}, margin={left,right,top,bottom},
                     size={largeur,hauteur,ratio}, bg="color", border=true/false }
-- construction du graphique g
    instructions graphiques en langage Lua ...
-- affichage du graphique g et sauvegarde dans le fichier <filename>.tkz
g:Show()
-- ou bien sauvegarde uniquement dans le fichier <filename>.tkz
g:Save()
```

```
\end{luadraw}
```

Sauvegarde du fichier *.tkz* : le graphique est exporté au format tikz dans un fichier (avec l'extension *tkz*), par défaut celui-ci est sauvegardé dans le dossier courant. Mais il est possible d'imposer un chemin spécifique en définissant dans le document, la commande `\luadrawTkzDir`, par exemple : `\def\luadrawTkzDir{tikz/}`, ce qui permettra d'enregistrer les fichiers **.tkz* dans le sous-dossier *tikz* du dossier courant, à condition toutefois que ce sous-dossier existe!

2) Options de l'environnement

Celles-ci sont :

- *name* = … : permet de donner un nom au fichier tikz produit, on donne un nom sans extension (celle-ci sera automatiquement ajoutée, c'est *.tkz*). Si cette option est omise, alors il y a un nom par défaut, qui est le nom du fichier maître suivi d'un numéro.
- *exec* = *true/false* : permet d'exécuter ou non le code Lua compris dans l'environnement. Par défaut cette option vaut true, **SAUF** si l'option globale *noexec* a été mentionnée dans le préambule avec la déclaration du paquet. Lorsqu'un graphique complexe qui demande beaucoup de calculs est au point, il peut être intéressant de lui ajouter l'option *exec=false*, cela évitera les recalculs de ce même graphique pour les compilations à venir.
- *auto* = *true/false* : permet d'inclure ou non automatiquement le fichier tikz en lieu et place de l'environnement *luadraw* lorsque l'option *exec* est à false. Par défaut l'option *auto* vaut true.

3) La classe cpx

Elle est automatiquement chargée par le module *graph_draw2d* et donc au chargement du paquet *luadraw*. Cette classe permet de manipuler les nombres complexes et de faire les calculs habituels. On crée un complexe avec la fonction **Z(a,b)** pour $a + i \times b$, ou bien avec la fonction **Zp(r,theta)** pour $r \times e^{i\theta}$ en coordonnées polaires.

- Exemple : *local z = Z(a,b)* va créer le complexe correspondant à $a + i \times b$ dans la variable *z*. On accède alors aux parties réelle et imaginaire de *z* comme ceci : *z.re* et *z.im*.
- **Attention** : un nombre réel *x* n'est pas considéré comme complexe par Lua. Cependant, les fonctions proposées pour la construction graphique font la vérification et la conversion réel vers complexe. On peut néanmoins, utiliser *Z(x,0)* à la place de *x*.
- Les opérateurs habituels ont été surchargés ce qui permet l'utilisation des symboles habituels, à savoir : +, x, -, /, ainsi que le test d'égalité avec =. Lorsqu'un calcul échoue le résultat renvoyé en principe doit être égal à *nil*.
- À cela s'ajoutent les fonctions suivantes (il faut utiliser la notation pointée en Lua) :
  - le module : **cpx.abs(z)**,
  - la norme 1 : **cpx.N1(z)**,
  - l'argument principal : **cpx.arg(z)**,
  - le conjugué : **cpx.bar(z)**,
  - l'exponentielle complexe : **cpx.exp(z)**,
  - le produit scalaire : **cpx.dot(z1,z2)**, où les complexes représentent des affixes de vecteurs,
  - le déterminant : **cpx.det(z1,z2)**,
  - l'arrondi : **cpx.round(z, nb decimales)**,
  - la fonction : **cpx.isNul(z)** teste si les parties réelle et imaginaire de *z* sont en valeur absolue inférieures à une variable *epsilon* qui vaut *1e-16* par défaut.

La dernière fonction renvoie un booléen, les fonctions bar, exponentielle et round renvoient un complexe, et les autres renvoient un réel.

On dispose également de la constante *cpx.I* qui représente l'imaginaire pur *i*.

Exemple :

```
1 local i = cpx.I
2 local A = 2+3*i
```

Le symbole de multiplication est obligatoire.

4) Création d'un graphe

Comme cela a été vu plus haut, la création se fait dans un environnement *luadraw*, c'est à la première ligne à l'intérieur de l'environnement qu'est faite cette création en nommant le graphique :

```
local g = graph:new{ window={x1,x2,y1,y2,xscale,yscale}, margin={left,right,top,bottom},
                     size={largeur,hauteur,ratio}, bg="color", border=true/false }
```

La classe *graph* est définie dans le paquet *luadraw*. On instancie cette classe en invoquant son constructeur et en donnant un nom (ici c'est *g*), on le fait en local de sorte que le graphique *g* ainsi créé, n'existera plus une fois sorti de l'environnement (sinon *g* resterait en mémoire jusqu'à la fin du document).

- Le paramètre (facultatif) *window* définit le pavé de $\mathbf{R}^2$ correspondant au graphique : c'est $[x_1,x_2]\times[y_1,y_2]$. Les paramètres *xscale* et *yscale* sont facultatifs et valent 1 par défaut, ils représentent l'échelle (cm par unité) sur les axes. Par défaut on a *window = {-5,5,-5,5,1,1}*.
- Le paramètre (facultatif) *margin* définit des marges autour du graphique en cm. Par défaut on a *margin = {0.5,0.5,0.5,0.5}*.
- Le paramètre (facultatif) *size* permet d'imposer une taille (en cm, marges incluses) pour le graphique, l'argument *ratio* correspond au rapport d'échelle souhaité (*xscale*/*yscale*), un ratio de 1 donnera un repère orthonormé, et si le ratio n'est pas précisé alors le ratio par défaut est conservé. L'utilisation de ce paramètre va modifier les valeurs de *xscale* et *yscale* pour avoir les bonnes tailles. Par défaut la taille est de 11×11 (en cm) avec les marges (10×10 sans les marges).
- Le paramètre (facultatif) *bg* permet de définir une couleur de fond pour le graphique, cette couleur est une chaîne de caractères représentant une couleur pour tikz. Par défaut cette chaîne est vide ce qui signifie que le fond ne sera pas peint.
- Le paramètre (facultatif) *border* indique si un cadre doit être dessiné ou non autour du graphique (en incluant les marges). Par défaut ce paramètre vaut false.

Construction du graphique.

- L'objet instancié (*g* ici dans l'exemple) possède un certain nombre de méthodes permettant de faire du dessin (segments, droites, courbes,...). Les instructions de dessins ne sont pas directement envoyées à TEX, elles sont enregistrées sous forme de chaînes dans une table qui est une propriété de l'objet *g*. C'est la méthode **g:Show()** qui va envoyer ces instructions à TEX tout en les sauvegardant dans un fichier texte[1]. La méthode **g:Save()** enregistre le graphique dans un fichier mais sans envoyer les instructions à TEX.
- Le paquet *luadraw* fournit aussi un certain nombre de fonctions mathématiques, ainsi que des fonctions permettant des calculs sur les listes (tables) de complexes, des transformations géométriques, ...etc.

Système de coordonnées. Repérage

- L'objet instancié (*g* ici dans l'exemple) possède :
  1. Une vue originelle : c'est le pavé de $\mathbf{R}^2$ défini par l'option *window* à la création. Celui-ci **ne doit pas être modifié** par la suite.
  2. Une vue courante : c'est un pavé de $\mathbf{R}^2$ qui doit être inclus dans la vue originelle, ce qui sort de ce pavé sera clippé. Par défaut la vue courante est la vue originelle. Pour retrouver la vue courante on peut utiliser la méthode **g:Getview()** qui renvoie une table `{x1,x2,y1,y2}`, celle-ci représente la pavé $[x1,x2]\times[y1,y2]$.
  3. Une matrice de transformation : celle-ci est initialisée à la matrice identité. Lors d'une instruction de dessin les points sont automatiquement transformés par cette matrice avant d'être envoyés à tikz.
  4. Un système de coordonnées (repère cartésien) lié à la vue courante, c'est le repère de l'utilisateur. Par défaut c'est le repère canonique de $\mathbf{R}^2$, mais il est possible d'en changer. Admettons que la vue courante soit le pavé $[-5,5]\times[-5,5]$, il est possible par exemple, de décider que ce pavé représente l'intervalle $[-1,12]$ pour les abscisses et l'intervalle $[0,8]$ pour les ordonnées, la méthode qui fait ce changement va modifier la matrice de transformation du graphe, de tel sorte que pour l'utilisateur tout se passe comme s'il était dans le pavé $[-1,12]\times[0,8]$. On peut retrouver les intervalles du repère de l'utilisateur avec les méthodes : **g:Xinf(), g:Xsup(), g:Yinf() et g:Ysup()**.
- On utilise les nombres complexes pour représenter les points ou les vecteurs dans le repère cartésien de l'utilisateur.

1. Ce fichier contiendra un environnement *tikzpicture*.

- Dans l'export tikz les coordonnées seront différentes car le coin inférieur gauche (hors marges) aura pour coordonnées $(0,0)$, et le coin supérieur droit (hors marges) aura des coordonnées correspondant à la taille (hors marges) du graphique, et avec 1 cm par unité sur les deux axes. Ce qui fait que normalement, tikz ne devrait manipuler que de « petits » nombres.
- La conversion se fait automatiquement avec la méthode **g:strCoord(x,y)** qui renvoie une chaîne de la forme *(a,b)*, où a et b sont les coordonnées pour tikz, ou bien avec la méthode **g:Coord(z)** qui renvoie aussi une chaîne de la forme *(a,b)* représentant les coordonnées tikz du point d'affixe z dans le repère de l'utilisateur.

5) Peut-on utiliser directement du tikz dans l'environnement *luadraw* ?

Supposons que l'on soit en train de créer un graphique nommé g dans un environnement *luadraw*. Il est possible d'écrire une instruction tikz lors de cette création, mais pas en utilisant `tex.sprint("<instruction tikz>")`, car alors cette instruction ne ferait pas partie du graphique g. Il faut pour cela utiliser la méthode **g:Writeln("<instruction tikz>")**, avec la contrainte que **les antislash doivent être doublés**, et sans oublier que les coordonnées graphiques d'un point dans g ne sont pas les mêmes pour tikz. Par exemple :

```
g:Writeln("\\draw"..g:Coord(Z(1,-1))..." node[red] {Texte};")
```

Ou encore pour changer des styles :

```
g:Writeln("\\tikzset{every node/.style={fill=white}}")
```

Dans une présentation beamer, cela peut aussi être utilisé pour insérer des pauses dans un graphique :

```
g:Writeln("\\pause")
```

II Méthodes graphiques

On peut créer des lignes polygonales, des courbes, des chemins, des points, des labels.

1) Lignes polygonales

Une ligne polygonale est une liste (table) de composantes connexes, et une composante connexe est une liste (table) de complexes qui représentent les affixes des points. Par exemple l'instruction :

```
local L = { {Z(-4,0), Z(0,2), Z(1,3)}, {Z(0,0), Z(4,-2), Z(1,-1)} }
```

crée une ligne polygonale à deux composantes dans une variable L.

Dessin d'une ligne polygonale. C'est la méthode **g:Dpolyline(L,close, draw_options)** (où g désigne le graphique en cours de création), L est une ligne polygonale, *close* un argument facultatif qui vaut *true* ou *false* indiquant si la ligne doit être refermée ou non (*false* par défaut), et *draw_options* est une chaîne de caractères qui sera passée directement à l'instruction *\draw* dans l'export.

Choix des options de dessin d'une ligne polygonale. On peut passer des options de dessin directement à l'instruction *\draw* dans l'export, mais elles auront un effet local uniquement. Il est possible de modifier ces options de manière globale avec la méthode **g:Lineoptions(style,color,width,arrows)** (lorsqu'un des arguments vaut *nil*, c'est sa valeur par défaut qui est utilisée) :

- *color* est une chaîne de caractères représentant une couleur connue de tikz ("black" par défaut),
- *style* est une chaîne de caractères représentant le type de ligne à dessiner, ce style peut être :
  - *"noline"* : trait non dessiné,
  - *"solid"* : trait plein, valeur par défaut,
  - *"dotted"* : trait pointillé,
  - *"dashed"* : tirets,
  - style personnalisé : l'argument *style* peut être une chaîne de la forme (exemple) *"{2.5pt}{2pt}"* ce qui signifie : un trait de 2.5pt suivi d'un espace de 2pt, le nombre de valeurs peut être supérieur à 2, ex : *"{2.5pt}{2pt}{1pt}{2pt}"* (succession de on, off).

- *width* est un nombre représentant l'épaisseur de ligne exprimée en dixième de points, par exemple 8 pour une épaisseur réelle de 0.8pt (valeur de 4 par défaut),
- *arrows* est une chaîne qui précise le type de flèche qui sera dessiné, cela peut être :
  - *"-"* qui signifie pas de flèche (valeur par défaut),
  - *"->"* qui signifie une flèche à la fin,
  - *"<-"* qui signifie une flèche au début,
  - *"<->"*qui signifie une flèche à chaque bout.

    ATTENTION : la flèche n'est pas dessinée lorsque l'argument *close* est true.

On peut modifier les options individuellement avec les méthodes :

- **g:Linecolor(color)**,
- **g:Linestyle(style)**,
- **g:Linewidth(width)**,
- **g:Arrows(arrows)**,
- plus les méthodes suivantes :
  - **g:Lineopacity(opacity)** qui règle l'opacité du tracé de la ligne, l'argument *opacity* doit être une valeur entre 0 (totalement transparent) et 1 (totalement opaque), par défaut la valeur est de 1.
  - **g:Linecap(style)**, pour jouer sur les extrémités de la ligne, l'argument *style* est une chaîne qui peut valoir :
    - *"butt"* (bout droit au point d'arrêt, valeur par défaut),
    - *"round"* (bout arrondi en demi-cercle),
    - *"square"* (bout « arrondi » en carré).
  - **g:Linejoin(style)**, pour jouer sur la jointure entre segments, l'argument style est une chaîne qui peut valoir :
    - *"miter"* (coin pointu, valeur par défaut),
    - *"round"* (ou coin arrondi),
    - *"bevel"* (coin coupé).

Options de remplissage d'une ligne polygonale. C'est la méthode **g:Filloptions(style,color,opacity,evenodd)** (qui utilise la librairie *patterns* de tikz, celle-ci est chargée avec le paquet). Lorsqu'un des arguments vaut *nil*, c'est sa valeur par défaut qui est utilisée :

- *color* est une chaîne de caractères représentant une couleur connue de tikz ("black" par défaut).
- *style* est une chaîne de caractères représentant le type de remplissage, ce style peut être :
  - *"none"* : pas de remplissage, c'est la valeur par défaut,
  - *"full"* : remplissage plein,
  - *"bdiag"* : hachures descendantes de gauche à droite,
  - *"fdiag"* : hachures montantes de gauche à droite,
  - *"horizontal"* : hachures horizontales,
  - *"vertical"* : hachures verticales,
  - *"hvcross"* : hachures horizontales et verticales,
  - *"diagcross"* : diagonales descendantes et montantes,
  - *"gradient"* : dans ce cas le remplissage se fait avec un gradient défini avec la méthode **g:Gradstyle(chaîne)**, ce style est passé tel quel à l'instruction *\draw*. Par défaut la chaîne définissant le style de gradient est *"left color = white, right color = red"*,
  - tout autre style connu de la librairie *patterns* est également possible.

On peut modifier certaines options individuellement avec les méthodes :

- **g:Fillopacity(opacity)**,
- **g:Filleo(evenodd)**.

```
\begin{luadraw}{name=champ}
local g = graph:new{window={-5,5,-5,5},bg="Cyan!30",size={7,7}}
local f = function(x,y) -- éq. diff. y'= 1-x*y^2=f(x,y)
    return 1-x*y^2
end
local A = Z(-1,1) -- A = -1+i
local deltaX, deltaY, long = 0.5, 0.5, 0.4
local champ = function(f)
    local vecteurs, v = {}
    for y = g:Yinf(), g:Ysup(), deltaY do
        for x = g:Xinf(), g:Xsup(), deltaX do
            v = Z(1,f(x,y)) -- coordonnées 1 et f(x,y)
            v = v/cpx.abs(v)*long -- normalisation de v
            table.insert(vecteurs, {Z(x,y), Z(x,y)+v} )
        end
    end
    return vecteurs -- vecteurs est une ligne polygonale
end
g:Daxes( {0,1,1}, {labelpos={"none","none"}, arrows="->"} )
g:Dpolyline( champ(f), "->,blue")
g:Dodesolve(f,A.re,A.im,
  {t={-2.75,5},draw_options="red,line width=0.8pt"})
g:Dlabeldot("$A$", A, {pos="S"})
g:Show()
\end{luadraw}
```

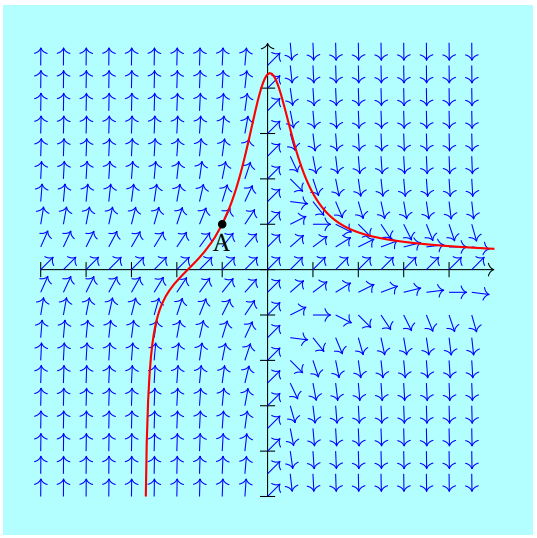

FIGURE 2 – Champ de vecteurs, courbe intégrale de $y' = 1 - xy^2$

2) Segments et droites

Un segment est une liste (table) de deux complexes représentant les extrémités. Une droite est une liste (table) de deux complexes, le premier représente un point de la droite, et le second un vecteur directeur de la droite (celui-ci doit être non nul).

2.1 Dangle

- La méthode **g:Dangle(B,A,C,r,draw_options)** dessine l'angle BAC avec un parallélogramme (deux côtés seulement sont dessinés), l'argument facultatif *r* précise la longueur d'un côté (0.25 par défaut). L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **angleD(B,A,C,r)** renvoie la liste des points de cet angle.

2.2 Dbissec

- La méthode **g:Dbissec(B,A,C,interior,draw_options)** dessine une bissectrice de l'angle géométrique BAC, intérieure si l'argument facultatif *interior* vaut *true* (valeur par défaut), extérieure sinon. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **bissec(B,A,C,interior)** renvoie cette bissectrice sous forme d'une liste *{A,u}* où *u* est un vecteur directeur de la droite.

2.3 Dhline

La méthode **g:Dhline(d,draw_options)** dessine une demi-droite, l'argument *d* doit être une liste de de complexes *{A,B}*, c'est la demi-droite [A, B) qui est dessinée.

Variante : **g:Dhline(A,B,draw_options)**. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

2.4 Dline

La méthode **g:Dline(d,draw_options)** trace la droite *d*, celle-ci est une liste du type *{A,u}* où *A* représente un point de la droite (un complexe) et *u* un vecteur directeur (un complexe non nul).

Variante : la méthode **g:Dline(A,B,draw_options)** trace la droite passant par les points A et B (deux complxes). L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

2.5 DlineEq

- La méthode **g:DlineEq(a,b,c,draw_options)** dessine la droite d'équation $ax + by + c = 0$. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **lineEq(a,b,c)** renvoie la droite d'équation $ax + by + c = 0$ sous la forme d'une liste *{A,u}* où A est un point de la droite et u un vecteur directeur.

2.6 Dmarkarc

La méthode **g:Dmarkarc(b,a,c,r,n,long,espace)** permet de marquer l'arc de cercle de centre a, de rayon r, allant de b à c, avec n petits segments. Par défaut la longueur (argument *long*) est de 0.25, et l'espacement (argument *espace*) et de 0.0625.

2.7 Dmarkseg

La méthode **g:Dmarkseg(a,b,n,long,espace,angle)** permet de marquer le segment défini par a et b avec n petits segments penchés de *angle* degrés (45° par défaut). Par défaut la longueur (argument *long*) est de 0.25, et l'espacement (argument *espace*) et de 0.125.

2.8 Dmed

- La méthode **g:Dmed(A,B, draw_options)** trace la médiatrice du segment [A; B].
  Variante : **g:Dmed(seg,draw_options)** où segment est une liste de deux points représentant le segment. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **med(A,B)** (ou **med(seg)**) renvoie la médiatrice du segment *[A,B]* sous la forme d'une liste *{C,u}* où C est un point de la droite et u un vecteur directeur.

2.9 Dparallel

- La méthode **g:Dparallel(d,A,draw_options)** trace la parallèle à d passant par A. L'argument d peut-être soit une droite (une liste constituée d'un point et un vecteur directeur) soit un vecteur non nul. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **parallel(d,A)** renvoie la parallèle à d passant par A sous la forme d'une liste *{B,u}* où B est un point de la droite et u un vecteur directeur.

2.10 Dperp

- La méthode **g:Dperp(d,A,draw_options)** trace la perpendiculaire à d passant par A. L'argument d peut-être soit une droite (une liste constituée d'un point et un vecteur directeur) soit un vecteur non nul. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **perp(d,A)** renvoie la perpendiculaire à d passant par A sous la forme d'une liste *{B,u}* où B est un point de la droite et u un vecteur directeur.

2.11 Dseg

La méthode **g:Dseg(seg,scale,draw_options)** dessine le segment défini par l'argument *seg* qui doit être une liste de deux complexes. L'argument facultatif *scale* (1 par défaut) est un nombre qui permet d'augmenter ou réduire la longueur du segment (la longueur naturelle est multipliée par *scale*). L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

2.12 Dtangent

- La méthode **g:Dtangent(p,t0,long,draw_options)** dessine la tangente à la courbe paramétrée par $p : t \mapsto p(t)$ (à valeurs complexes), au point de paramètre $t0$. Si l'argument *long* vaut *nil* (valeur par défaut) alors c'est la droite

entière qui est dessinée, sinon c'est un segment de longueur *long*. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

- La fonction **tangent(p,t0,long)** renvoie soit la droite, soit un segment (suivant que *long* vaut *nil* ou pas).

2.13 DtangentC

- La méthode **g:DtangentC(f,x0,long,draw_options)** dessine la tangente à la courbe cartésienne d'équation $y = f(x)$, au point d'abscisse *x0*. Si l'argument *long* vaut *nil* (valeur par défaut) alors c'est la droite entière qui est dessinée, sinon c'est un segment de longueur *long*. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **tangentC(f,x0,long)** renvoie soit la droite, soit un segment (suivant que *long* vaut *nil* ou pas).

```
\begin{luadraw}{name=orthocentre}
local g = graph:new{window={-5,5,-5,5},bg="cyan!30",size={7,7}}
local i = cpx.I
local A, B, C = 4*i, -2-2*i, 3.5
local h1, h2 = perp({B,C-B},A), perp({A,B-A},C) -- hauteurs
local A1, F = proj(A,{B,C-B}), proj(C,{A,B-A}) -- projetés
local H = interD(h1,h2) -- orthocentre
local A2 = 2*A1-H -- symétrique de H par rapport à BC
g:Dpolyline({A,B,C},true,
  "draw=none,fill=Maroon,fill opacity=0.3") -- fond du triangle
g:Linewidth(6); g:Filloptions("full", "blue", 0.2)
g:Dangle(C,A1,A,0.25); g:Dangle(B,F,C,0.25) -- angles droits
g:Linecolor("black"); g:Filloptions("full","cyan",0.5)
g:Darc(H,C,A2,1); g:Darc(B,A,A1,1) -- arcs
g:Filloptions("none","black",1) -- on rétablit l'opacité à 1
g:Dmarkarc(H,C,A1,1,2); g:Dmarkarc(A1,C,A2,1,2) -- marques
g:Dmarkarc(B,A,H,1,2)
g:Linewidth(8); g:Linecolor("black")
g:Dseg({A,B},1.25); g:Dseg({C,B},1.25); g:Dseg({A,C},1.25) -- côtés
g:Linecolor("red"); g:Dcircle(A,B,C) -- cercle
g:Linecolor("blue"); g:Dline(h1); g:Dline(h2) -- hauteurs
g:Dseg({A2,C}); g:Linecolor("red"); g:Dseg({H,A2}) -- segments
g:Dmarkseg(H,A1,2); g:Dmarkseg(A1,A2,2) -- marques
g:Labelcolor("blue") -- pour les labels
g:Dlabel("$A$",A, {pos="NW",dist=0.1}, "$B$",B, {pos="SW"},
    "$A_2$",A2,{pos="E"}, "$C$", C, {pos="S" },
    "$H$", H, {pos="NE"}, "$A_1$", A1, {pos="SW"})
g:Linecolor("black"); g:Filloptions("full")
g:Ddots({A,B,C,H,A1,A2}) -- dessin des points
g:Show(true)
\end{luadraw}
```

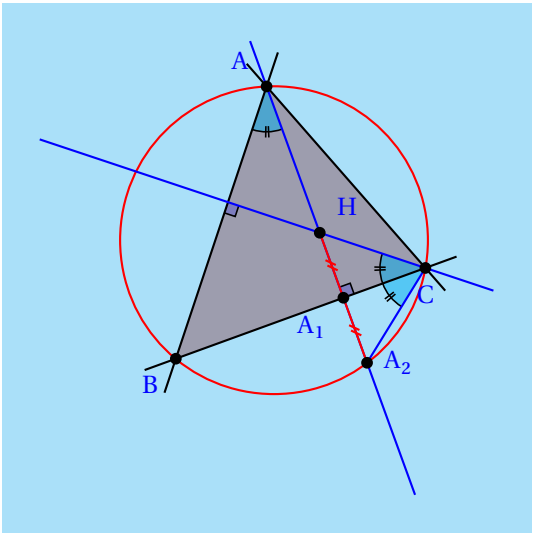

FIGURE 3 – Symétrique de l'orthocentre

3) Figures géométriques

3.1 Darc

- La méthode **g:Darc(B,A,C,r,sens,draw_options)** dessine un arc de cercle de centre *A* (complexe), de rayon *r*, allant de *B* (complexe) vers *C* (complexe) dans le sens trigonométrique si l'argument *sens* vaut 1, le sens inverse sinon. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **arc(B,A,C,r,sens)** renvoie la liste des points de cet arc (ligne polygonale).
- La fonction **arcb(B,A,C,r,sens)** renvoie cet arc sous forme d'un chemin (voir Dpath) utilisant des courbes de Bézier.

3.2 Dcircle

- La méthode **g:Dcircle(c,r,d,draw_options)** trace un cercle. Lorsque l'argument *d* est nil, c'est le cercle de centre *c* (complexe) et de rayon *r*, lorsque *d* est précisé (complexe) alors c'est le cercle passant par les points d'affixe *c*,*r* et *d*. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

- La fonction **circle(c,r,d)** renvoie la liste des points de ce cercle (ligne polygonale).
- La fonction **circleb(c,r,d)** renvoie ce cercle sous forme d'un chemin (voir Dpath) utilisant des courbes de Bézier.

3.3 Dellipse

- La méthode **g:Dellipse(c,rx,ry,inclin,draw_options)** dessine l'ellipse de centre *c* (complexe), les arguments *rx* et *ry* précisent les deux rayons (sur x et sur y), l'argument facultatif *inclin* est un angle en degrés qui indique l'inclinaison de l'ellipse par rapport à l'axe O*x* (angle nul par défaut). L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ellipse(c,rx,ry,inclin)** renvoie la liste des points de cette ellipse (ligne polygonale).
- La fonction **ellipseb(c,rx,ry,inclin)** renvoie cette ellipse sous forme d'un chemin (voir Dpath) utilisant des courbes de Bézier.

3.4 Dellipticarc

- La méthode **g:Dellipticarc(B,A,C,rx,ry,sens,inclin,draw_options)** dessine un arc d'ellipse de centre *A* (complexe) de rayons *rx* et *ry*, faisant un angle égal à *inclin* par rapport à l'axe O*x* (angle nul par défaut), allant de *B* (complexe) vers *A* (complexe) dans le sens trigonométrique si l'argument *sens* vaut 1, le sens inverse sinon. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **ellipticarc(B,A,C,rx,ry,sens,inclin)** renvoie la liste des points de cet arc (ligne polygonale).
- La fonction **ellipticarcb(B,A,C,rx,ry,sens,inclin)** renvoie cet arc sous forme d'un chemin (voir Dpath) utilisant des courbes de Bézier.

3.5 Dpolyreg

- La méthode **g:Dpolyreg(sommet1,sommet2,nbcotes,sens,draw_options)** ou **g:Dpolyreg(centre,sommet,nbcotes,draw_options)** dessine un polygone régulier. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **polyreg(sommet1,sommet2,nbcotes,sens)** et la fonction **polyreg(centre,sommet,nbcotes)**, renvoient la liste des sommets de ce polygone régulier.

3.6 Drectangle

- La méthode **g:Drectangle(a,b,c,draw_options)** dessine le rectangle ayant comme sommets consécutif *a* et *b* et dont le côté opposé passe par *c*. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **rectangle(a,b,c)** renvoie la liste des sommets de ce rectangle.

3.7 Dsequence

- La méthode **g:Dsequence(f,u0,n,draw_options)** fait le dessin des "escaliers" de la suite récurrente définie par son premier terme *u0* et la relation $u_{k+1} = f(u_k)$. L'argument f doit être une fonction d'une variable réelle et à valeurs réelles, l'argument *n* est le nombre de termes calculés. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **sequence(f,u0,n)** renvoie la liste des points constituant ces "escaliers".

```
\begin{luadraw}{name=sequence}
local g = graph:new{window={-0.1,1.7,-0.1,1.1},size={7,7,0}}
local i, pi, cos = cpx.I, math.pi, math.cos
local f = function(x) return cos(x)-x end
local ell = solve(f,0,pi/2)[1]
local L = sequence(cos,0.2,5) -- u_{n+1}=cos(u_n), u_0=0.2
local seg, z = {}, L[1]
for k = 2, #L do
    table.insert(seg,{z,L[k]})
    z = L[k]
end -- seg est la liste des segments de l'escalier
g:Writeln("\\tikzset{->-/.style={decoration={markings, mark=at
 ↪  position #1 with {\\arrow{Stealth}}}, postaction={decorate}}}")
g:Daxes({0,1,1}, {arrows="-Stealth"})
g:DlineEq(1,-1,0,"line width=0.8pt,ForestGreen")
g:Dcartesian(cos, {x={0,pi/2},draw_options="line
 ↪  width=1.2pt,Crimson"})
g:Dpolyline(seg,false,"->-=0.65,blue")
g:Dlabel("$u_0$",0.2,{pos="S",node_options="blue"})
g:Dseg({ell, ell*(1+i)},1,"dashed,gray")
g:Dlabel("$\\ell\\approx"..round(ell,3).."$", ell,{pos="S"})
g:Ddots(ell*(1+i)); g:Labelcolor("Crimson")
g:Dlabel("${\\mathcal C}_{\\cos}$",Z(1,cos(1)),{pos="E"})
g:Labelcolor("ForestGreen"); g:Labelangle(g:Arg(1+i)*180/pi)
g:Dlabel("$y=x$",Z(0.4,0.4),{pos="S",dist=0.1})
g:Show()
\end{luadraw}
```

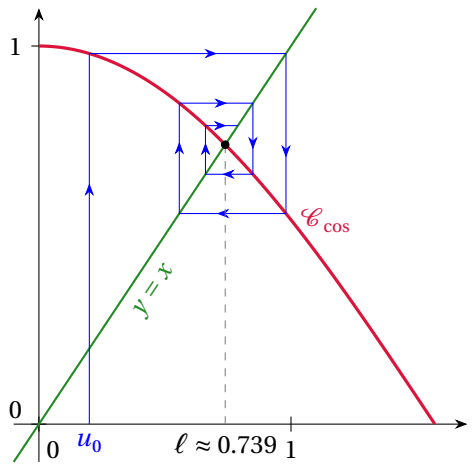

FIGURE 4 – Suite $u_{n+1} = \cos(u_n)$

La méthode **g:Arg(z)** calcule et renvoie l'argument *réel* du complexe z, c'est à dire son argument (en radians) à l'export dans le repère de tikz (il faut pour cela appliquer la matrice de transformation du graphe à z, puis faire le changement de repère vers celui de tikz). Si le repère du graphe est orthonormé et si la matrice de transformation est l'identité alors le résultat est identique à celui de **cpx.arg(z)** (ce n'est pas le cas dans l'exemple ci-dessus).

De même, la méthode **g:Abs(z)** calcule et renvoie le module *réel* du complexe z, c'est à dire son module à l'export dans le repère de tikz, c'est donc une longueur en centimètres. Si le repère du graphe est orthonormé avec 1cm par unité sur chaque axe, et si la matrice de transformation est une isométrie alors le résultat est identique à celui de **cpx.abs(z)**.

3.8 Dsquare

- La méthode **g:Dsquare(a,b,sens,draw_options)** dessine le carré ayant comme sommets consécutifs a et b, dans le sens trigonométrique lorsque *sens* vaut 1 (valeur par défaut). L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **square(a,b,sens)** renvoie la liste des sommets de ce carré.

3.9 Dwedge

La méthode **g:Dwedge(B,A,C,r,sens,draw_options)** dessine un secteur angulaire de centre A (complexe), de rayon r, allant de B (complexe) vers C (complexe) dans le sens trigonométrique si l'argument *sens* vaut 1, le sens inverse sinon. L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

4) Courbes

4.1 Paramétriques : Dparametric

- La fonction **parametric(p,t1,t2,nbdots,discont,nbdiv)** fait le calcul des points et renvoie une ligne polygonale (pas de dessin).
  - L'argument *p* est le paramétrage, ce doit être une fonction d'une variable réelle t et à valeurs complexes, par exemple : `local p = function(t) return cpx.exp(t*cpx.I) end`
  - Les arguments *t1* et *t2* sont obligatoires avec $t1 < t2$, ils forment les bornes de l'intervalle pour le paramètre.

- L'argument *nbdots* est facultatif, c'est le nombre de points (minimal) à calculer, il vaut 50 par défaut.
- L'argument *discont* est un booléen facultatif qui indique s'il y a des discontinuités ou non, c'est *false* par défaut.
- L'argument *nbdiv* est un entier positif qui vaut 5 par défaut et indique le nombre de fois que l'intervalle entre deux valeurs consécutives du paramètre peut être coupé en deux (dichotomie) lorsque les points correspondants sont trop éloignés.

- La méthode **g:Dparametric(p,args)** fait le calcul des points et le dessin de la courbe paramétrée par *p*. Le paramètre *args* est une table à 5 champs :

```
{ t={t1,t2}, nbdots=50, discont=true/false, nbdiv=5, draw_options="" }
```

  - Par défaut, le champ *t* est égal à *{g:Xinf(),g:Xsup()}*,
  - le champ *nbdots* vaut 50,
  - le champ *discont* vaut *false*,
  - le champ *nbdiv* vaut 5,
  - le champ *draw_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction *\draw*).

4.2 Polaires : Dpolar

- La fonction **polar(rho,t1,t2,nbdots,discont,nbdiv)** fait le calcul des points et renvoie une ligne polygonale (pas de dessin). L'argument *rho* est le paramétrage polaire de la courbe, ce doit être une fonction d'une variable réelle *t* et à valeurs réelles, par exemple :
  ```
  local rho = function(t) return 4*math.cos(2*t) end
  ```
  Les autres arguments sont identiques aux courbes paramétrées.
- La méthode **g:Dpolar(rho,args)** fait le calcul des points et le dessin de la courbe polaire paramétrée par *rho*. Le paramètre *args* est une table à 5 champs :

```
{ t={t1,t2}, nbdots=50, discont=true/false, nbdiv=5, draw_options="" }
```

  - Par défaut, le champ *t* est égal à $\{-\pi,\pi\}$,
  - le champ *nbdots* vaut 50,
  - le champ *discont* vaut *false*,
  - le champ *nbdiv* vaut 5,
  - le champ *draw_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction *\draw*).

4.3 Cartésiennes : Dcartesian

- La fonction **cartesian(f,x1,x2,nbdots,discont,nbdiv)** fait le calcul des points et renvoie une ligne polygonale (pas de dessin). L'argument *f* est la fonction dont on veut la courbe, ce doit être une fonction d'une variable réelle *x* et à valeurs réelles, par exemple :
  ```
  local f = function(x) return 1+3*math.sin(x)*x end
  ```
  Les arguments *x1* et *x2* sont obligatoires et forment les bornes de l'intervalle pour la variable. Les autres arguments sont identiques aux courbes paramétrées.
- La méthode **g:Dcartesian(f,args)** fait le calcul des points et le dessin de la courbe de *f*. Le paramètre *args* est une table à 5 champs :

```
{ x={x1,x2}, nbdots=50, discont=true/false, nbdiv=5, draw_options="" }
```

  - Par défaut, le champ *x* est égal à *{g:Xinf(),g:Xsup()}*,
  - le champ *nbdots* vaut 50,
  - le champ *discont* vaut *false*,
  - le champ *nbdiv* vaut 5,
  - le champ *draw_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction *\draw*).

4.4 Fonctions périodiques : Dperiodic

- La fonction **periodic(f,period,x1,x2,nbdots,discont,nbdiv)** fait le calcul des points et renvoie une ligne polygonale (pas de dessin).
  - L'argument *f* est la fonction dont on veut la courbe, ce doit être une fonction d'une variable réelle *x* et à valeurs réelles.

- L'argument *period* est une table du type *{a,b}* avec $a < b$ représentant une période de la fonction *f*.
- Les arguments *x1* et *x2* sont obligatoires et forment les bornes de l'intervalle pour la variable.
- Les autres arguments sont identiques aux courbes paramétrées.

- La méthode **g:Dperiodic(f,period,args)** fait le calcul des points et le dessin de la courbe de *f*. Le paramètre *args* est une table à 5 champs :

```
{ x={x1,x2}, nbdots=50, discont=true/false, nbdiv=5, draw_options="" }
```

  - Par défaut, le champ *x* est égal à *{g:Xinf(),g:Xsup()}*,
  - le champ *nbdots* vaut 50,
  - le champ *discont* vaut *false*,
  - le champ *nbdiv* vaut 5,
  - le champ *draw_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction *\draw*).

4.5 Fonctions en escaliers : Dstepfunction

- La fonction **stepfunction(def,discont)** fait le calcul des points et renvoie une ligne polygonale (pas de dessin).
  - L'argument *def* permet de définir la fonction en escaliers, c'est une table à deux champs :

    ```
    { {x1,x2,x3,...,xn}, {c1,c2,...} }
    ```

    Le premier élément *{x1,x2,x3,...,xn}* doit être une subdivision du segment $[x1, xn]$.
    Le deuxième élément *{c1,c2,...}* est la liste des constantes avec *c1* pour le morceau *[x1,x2]*, *c2* pour le morceau *[x2,x3]*, etc.
  - L'argument *discont* est un booléen qui vaut *true* par défaut.
- La méthode **g:Dstepfunction(def,args)** fait le calcul des points et le dessin de la courbe de la fonction en escalier.
  - L'argument *def* est le même que celui décrit au-dessus (définition de la fonction en escalier).
  - L'argument *args* est une table à 2 champs :

    ```
    { discont=true/false, draw_options="" }
    ```

    Par défaut, le champ *discont* vaut true, et le champ *draw_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction *\draw*).

4.6 Fonctions affines par morceaux : Daffinebypiece

- La fonction **affinebypiece(def,discont)** fait le calcul des points et renvoie une ligne polygonale (pas de dessin).
  - L'argument *def* permet de définir la fonction en escaliers, c'est une table à deux champs :

    ```
    { {x1,x2,x3,...,xn}, { {a1,b1}, {a2,b2},...} }
    ```

    Le premier élément *{x1,x2,x3,...,xn}* doit être une subdivision du segment $[x1, xn]$.
    Le deuxième élément *{ {a1,b1},{a2,b2},...}* signifie que sur *[x1,x2]* la fonction est $x \mapsto a_1 x + b_1$, sur *[x2,x3]* la fonction est $x \mapsto a_2 x + b_2$, etc.
  - L'argument *discont* est un booléen qui vaut *true* par défaut.
- La méthode **g:Daffinebypiece(def,args)** fait le calcul des points et le dessin de la courbe de la fonction affine par morceaux.
  - L'argument *def* est le même que celui décrit au-dessus (définition de la fonction affine par morceaux).
  - L'argument *args* est une table à 2 champs :

    ```
    { discont=true/false, draw_options="" }
    ```

    Par défaut, le champ *discont* vaut *true*, et le champ *draw_options* est une chaîne vide (celle-ci sera transmise telle quelle à l'instruction *\draw*).

4.7 Équations différentielles : Dodesolve

- La fonction **odesolve(f,t0,Y0,tmin,tmax,nbdots,method)** permet une résolution approchée de l'équation différentielle $Y'(t) = f(t, Y(t))$ dans l'intervalle [tmin,tmax] qui doit contenir *t0*, avec la condition initiale $Y(t0) = Y0$.
  - L'argument f est une fonction $f : (t, Y) -> f(t, Y)$ à valeurs dans R^n et où Y est également dans R^n : *Y={y1, y2,..., yn}* (lorsque $n = 1$, Y est un réel).
  - Les arguments *t0* et *Y0* donnent les conditions initiales avec *Y0={y1(t0), ..., yn(t0)}* (les yi sont réels), ou *Y0=y1(t0)* lorsque $n = 1$.

– Les arguments *tmin* et *tmax* définissent l'intervalle de résolution, celui-ci doit contenir $t0$.
– L'argument *nbdots* indique le nombre de points calculés de part et d'autre de $t0$.
– L'argument optionnel *method* est une chaîne qui peut valoir *"rkf45"* (valeur par défaut), ou *"rk4"*. Dans le premier cas, on utilise la méthode de Runge Kutta-Fehlberg (à pas variable), dans le second cas c'est la méthode classique de Runge-Kutta d'ordre 4.
– En sortie, la fonction renvoie la matrice suivante (liste de listes de réels) :

```
{ {tmin,...,tmax}, {y1(tmin),...,y1(tmax)}, {y2(tmin),...,y2(tmax)},...}
```

La première composante est la liste des valeurs de t (dans l'ordre croissant), la deuxième est la liste des valeurs (approchées) de la composante *y1* correspondant à ces valeurs de *t*, ... etc.

- La méthode **g:DplotXY(X,Y,draw_options)**, où les arguments X et Y sont deux listes de réels de même longueur, dessine la ligne polygonale constituée des points (X[k],Y[k]). L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

```
\begin{luadraw}{name=lokta_volterra}
local g = graph:new{window={-5,50,-0.5,5},size={7,10,0},
 ↪  border=true}
local i = cpx.I
local f = function(t,y) return {y[1]-y[1]*y[2],-y[2]+y[1]*y[2]} end
g:Labelsize("footnotesize")
g:Daxes({0,10,1},{limits={{0,50},{0,4}}, nbsubdiv={4,0},
 ↪  legendsep={0.1,0}, originpos={"center","center"},
 ↪  legend={"$t$",""}})
local y0 = {2,2}
local M = odesolve(f,0,y0,0,50,250) -- résolution approchée
-- M est une table à 3 éléments: t, x et y
g:Lineoptions("solid","blue",8)
g:Dseg({5+3.5*i,10+3.5*i}); g:Dlabel("$x$",10+3.5*i,{pos="E"})
g:DplotXY(M[1],M[2]) -- points (t,x(t))
g:Linecolor("red")
g:Dseg({5+3*i,10+3*i}); g:Dlabel("$y$",10+3*i,{pos="E"})
g:DplotXY(M[1],M[3])  -- points (t,y(t))
g:Lineoptions(nil,"black",4)
g:Saveattr(); g:Viewport(20,50,3,5) -- changement de vue
g:Coordsystem(-0.5,3.25,-0.5,3.25) -- nouveau repère associé
g:Daxes({0,1,1},{legend={"$x$","$y$"},arrows="->"})
g:Lineoptions(nil,"ForestGreen",8)
g:DplotXY(M[2],M[3]) -- points (x(t),y(t))
g:Restoreattr() -- retour à l'ancienne vue
g:Dlabel("$\\begin{cases}x'=x-xy\\\\y'=-y+xy\\end{cases}$",
 ↪  5+4.75*i,{})
g:Show()
\end{luadraw}
```

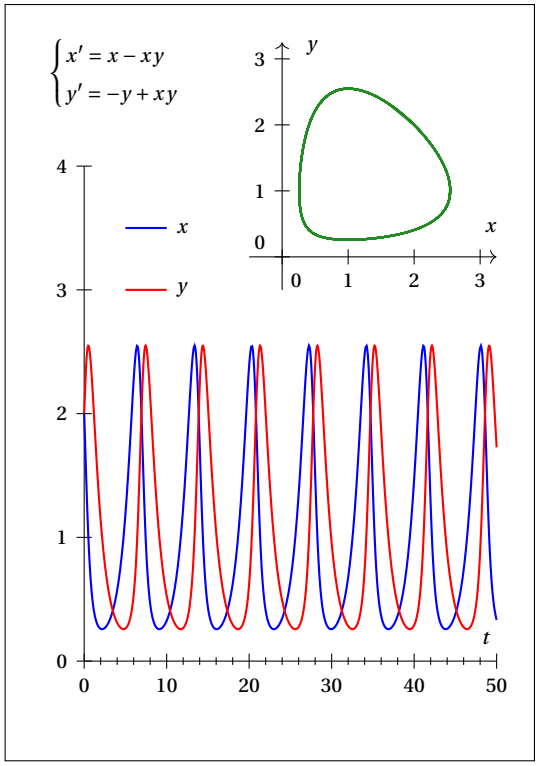

FIGURE 5 – Un système différentiel de Lokta-Volterra

- La méthode **g:Dodesolve(f,t0,Y0,args)** permet le dessin d'une solution à l'équation $Y'(t) = f(t, Y(t))$.
  – L'argument obligatoire f est une fonction $f : (t, Y) -> f(t, Y)$ à valeurs dans R^n et où Y est également dans R^n : *Y={y1, y2,..., yn}* (lorsque $n = 1$, Y est un réel).
  – Les arguments *t0* et *Y0* donnent les conditions initiales avec *Y0={y1(t0), ..., yn(t0)}* (les yi sont réels), ou *Y0=y1(t0)* lorsque $n = 1$.
  – L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 5 champs :

```
{ t={tmin,tmax}, out={i1,i2}, nbdots=50, method="rkf45"/"rk4", draw_options="" }
```

  * Le champ *t* détermine l'intervalle pour la variable *t*, par défaut il vaut *{g:Xinf(), g:Xsup()}*.
  * Le champ *out* est une table de deux entiers {i1, i2}, si *M* désigne la matrice renvoyée par la fonction *odesolve*, les points dessinés auront pour abscisses les M[i1] et pour ordonnées les M[i2]. Par défaut on a *i1=1* et *i2=2*, ce qui correspond à la fonction *y1* en fonction de *t*.
  * Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
  * Le champ *method* détermine la méthode à utiliser, les valeurs possibles sont *"rkf45"* (valeur par défaut), ou *"rk4"*.
  * Le champ *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

4.8 Courbes implicites : Dimplicit

- La fonction **implicit(f,x1,x2,y1,y2,grid)** calcule et renvoie une une ligne polygonale constituant la courbe implicite d'équation $f(x,y)=0$ dans le pavé $[x_1,x_2]\times[y_1,y_2]$. Ce pavé est découpé en fonction du paramètre *grid*.
  - L'argument obligatoire f est une fonction $f:(x,y)->f(x,y)$ à valeurs dans R.
  - Les arguments *x1, x2 ,y1, y2* définissent la fenêtre du tracé, qui sera le pavé $[x_1,x_2]\times[y_1,y_2]$, on doit avoir $x1<x2$ et $y1<y2$.
  - L'argument *grid* est une table contenant deux entiers positifs : {n1,n2}, le premier entier indique le nombre de subdivisions suivant x, et le second le nombre de subdivisions suivant y.
- La méthode **g:Dimplicit(f,args)** fait le dessin de la courbe implicite d'équations $f(x,y)=0$.
  - L'argument obligatoire f est une fonction $f:(x,y)->f(x,y)$ à valeurs dans R.
  - L'argument *args* permet de définir les paramètres du tracé, c'est une table à 3 champs :

```
{ view={x1,x2,y1,y2}, grid={n1,n2}, draw_options="" }
```

    * Le champ *view* détermine la zone de dessin $[x_1,x_2]\times[y_1,y_2]$. Par défaut on a *view={g:Xinf(), g:Xsup(), g:Yinf(), g:Ysup()}*,
    * le champ *grid* détermine la grille, ce champ vaut par défaut *{50,50}*,
    * le champ *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

4.9 Courbes de niveau : Dcontour

La méthode **g:Dcontour(f,z,args)** fait le dessin de **lignes de niveau** de la fonction $f:(x,y)->f(x,y)$ à valeurs réelles.

- L'argument z (obligatoire) est la liste des différents niveaux à tracer.
- L'argument *args* (facultatif) permet de définir les paramètres du tracé, c'est une table à 4 champs :

```
{ view={x1,x2,y1,y2}, grid={n1,n2}, colors={"color1","color2",...}, draw_options="" }
```

  - Le champ *view* détermine la zone de dessin [x1,x2]x[y1,y2], par défaut on a *view={g:Xinf(),g:Xsup(), g:Yinf(), g:Ysup()}*.
  - Le champ *grid* détermine la grille, par défaut on a *grid={50,50}*.
  - Le champ *colors* est la liste des couleurs par niveau, par défaut cette liste est vide et c'est la couleur courante de tracé qui est utilisée.
  - Le champ *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

```
\begin{luadraw}{name=Dcontour}
local g = graph:new{window={-1,6.5,-1.5,11},size={7,7,0}}
local i, sin, cos = cpx.I, math.sin, math.cos
local f = function(x,y) return (x+y)/(2+cos(x)*sin(y)) end
local rainbow = {Purple,Indigo,Blue,Green,Yellow,Orange,Red}
local Lz = range(1,10) -- niveaux à tracer
local Colors = {} -- liste des couleurs une par niveau
for k = 1,10 do
    table.insert(Colors, palette(rainbow,k/10))
end
g:Dgradbox({0,5+10*i,1,1},{legend={"$x$","$y$"},grid=true,
  title="$z=\\frac{x+y}{2+\\cos(x)\\sin(y)}$"})
g:Linewidth(12)
g:Dcontour(f,Lz,{view={0,5,0,10}, colors=Colors})
for k = 1, 10 do
    local y = (2*k+4)/3*i
    g:Dseg({5.25+y,5.5+y},1,"color="..Colors[k])
    g:Labelcolor(Colors[k])
    g:Dlabel("$z="..k.."$",5.5+y,{pos="E"})
end
g:Show()
\end{luadraw}
```

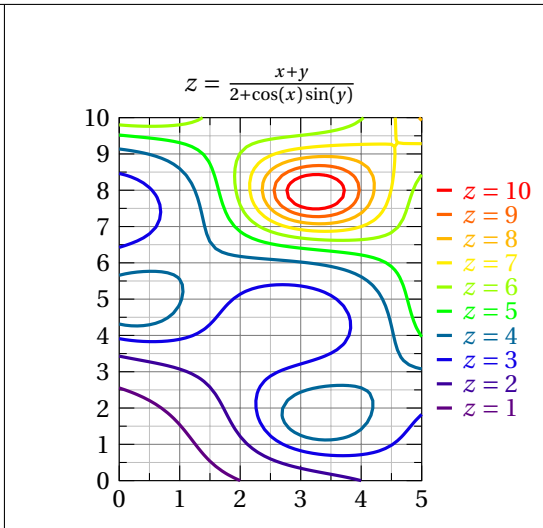

FIGURE 6 – Exemple avec Dcontour

5) Domaines liés à des courbes cartésiennes

5.1 Ddomain1

La méthode **g:Ddomain1(f,args)** dessine le contour délimité par la courbe de la fonction f sur un intervalle $[a;b]$, l'axe Ox, et les droites $x=a$, $x=b$.

L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 5 champs :

```
{ x={a,b},  nbdots=50, discont=false, nbdiv=5, draw_options="" }
```

- Le champ *x* détermine l'intervalle d'étude, par défaut il vaut *{g:Xinf(), g:Xsup()}*.
- Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
- Le champ *discont* indique s'il y a ou non des discontinuité pour la fonction (*false* par défaut).
- Le champ *nbdiv* est utilisé dans la méthode de calcul des points de la courbe (5 par défaut).
- Le champ *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

5.2 Ddomain2

La méthode **g:Ddomain2(f,g,args)** dessine le contour délimité par la courbe de la fonction f et la courbe de la fonction g sur un intervalle $[a;b]$.

L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 6 champs :

```
{ x={a,b}, nbdots=50, discont=false, nbdiv=5, draw_options="" }
```

- Le champ *x* détermine l'intervalle d'étude, par défaut il vaut *{g:Xinf(), g:Xsup()}*.
- Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
- Le champ *discont* indique s'il y a ou non des discontinuité pour la fonction (false par défaut).
- Le champ *nbdiv* est utilisé dans la méthode de calcul des points de la courbe (5 par défaut).
- Le champ *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

5.3 Ddomain3

La méthode **g:Ddomain3(f,g,args)** dessine le contour délimité par la courbe de la fonction f et celle de la fonction g.

L'argument *args* (facultatif) permet de définir les paramètres pour la courbe, c'est une table à 5 champs :

```
{ x={a,b}, nbdots=50, discont=false, nbdiv=5, draw_options="" }
```

- Le champ *x* détermine l'intervalle d'étude, par défaut il vaut *{g:Xinf(), g:Xsup()}*.
- Le champ *nbdots* détermine le nombre de points à calculer pour la fonction (50 par défaut).
- Le champ *discont* indique s'il y a ou non des discontinuité pour la fonction (false par défaut).
- Le champ *nbdiv* est utilisé dans la méthode de calcul des points de la courbe (5 par défaut).
- Le champ *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.

```
\begin{luadraw}{name=courbe}
local g = graph:new{ window={-5,5,-5,5}, bg="", size={7,7} }
local f = function(x) return (x-2)^2-2 end
local h = function(x) return 2*math.cos(x-2.5)-2.25 end
g:Daxes( {0,1,1},{grid=true,gridstyle="dashed",
                  arrows="->"})
g:Filloptions("full","brown",0.3)
g:Ddomain1( math.floor, { x={-2.5,3.5} })
g:Filloptions("none","white",1); g:Lineoptions("solid","red",12)
g:Dstepfunction( {range(-5,5), range(-5,4)},{draw_options=
  "arrows={Bracket-Bracket[reversed]},shorten >=-2pt"})
g:Labelcolor("red")
g:Dlabel("Partie entière",Z(-3,3),{node_options="fill=white"})
g:Ddomain3(f,h,{draw_options="fill=blue,fill opacity=0.6"})
g:Dcartesian(f, {x={0,5}, draw_options="blue"})
g:Dcartesian(h, {x={0,5}, draw_options="green"})
g:Show()
\end{luadraw}
```

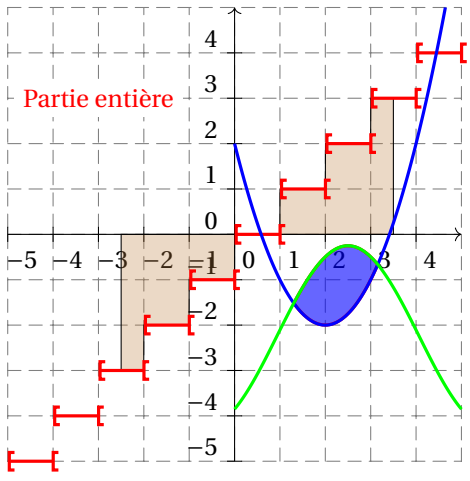

FIGURE 7 – Partie entière, fonctions Ddomain1 et Ddomain3

6) Points (Ddots) et labels (Dlabel)

- La méthode pour dessiner un ou plusieurs points est : **g:Ddots(dots, mark_options)**.
  - L'argument *dots* peut être soit un seul point (donc un complexe), soit une liste (une table) de complexes, soit une liste de liste de complexes. Les points sont dessinés dans la couleur courante du tracé de lignes.
  - L'argument *mark_options* est une chaîne de caractères facultative qui sera passée telle quelle à l'instruction *\draw* (modifications locales), exemple :
    ```
    "color=green, line width=1.2, scale=0.25"
    ```
  - Deux méthodes pour modifier globalement l'apparence des points :
    * La méthode **g:Dotstyle(style)** qui définit le style de point, l'argument *style* est une chaîne de caractères qui vaut par défaut *"*"*. Les styles possibles sont ceux de la librairie *plotmarks*.
    * La méthode **g:Dotscale(scale)** permet de jouer sur la taille du point, l'argument *scale* est un entier positif qui vaut 1 par défaut, il sert à multiplier la taille par défaut du point. La largeur courante de tracé de ligne intervient également dans la taille du point. Pour les style de points "pleins" (par exemple le style *triangle**), le style et la couleur de remplissage courants sont utilisés par la librairie.
- La méthode pour placer un label est :

  g:Dlabel(text1, anchor1, args1, text2, anchor2, args2, ...).
  - Les arguments *text1, text2,...* sont des chaînes de caractères, ce sont les labels.
  - Les arguments *anchor1, anchor2,...* sont des complexes représentant les points d'ancrage des labels.
  - Les arguments *args1,arg2,...* permettent de définir localement les paramètres des labels, ce sont des tables à 3 champs :
    ```
    { pos=nil, dist=0, node_options="" }
    ```
    * Le champ *pos* indique la position du label par rapport au point d'ancrage, il peut valoir *"N"* pour nord, *"NE"* pour nord-est, *"NW"* pour nord-ouest, ou encore *"S"*, *"SE"*, *"SW"*. Par défaut, il vaut *center*, et dans ce cas le label est centré sur le point d'ancrage.
    * Le champ *dist* est une distance en cm qui vaut 0 par défaut, c'est la distance entre le label et son point d'ancrage lorsque *pos* n'est pas égal a *center*.
    * L'argument *node_options* est une chaîne (vide par défaut) destinée à recevoir des options qui seront directement passées à tikz dans l'instruction *node[]*.
    * Les labels sont dessinés dans la couleur courante du texte du document, mais on peut changer de couleur avec l'argument *node_options* en mettant par exemple : *node_options="color=blue"*.

      Attention : les options choisies pour un label s'appliquent aussi aux labels suivants si elles sont inchangées.

  Options globales pour les labels :
  - la méthode **g:Labelstyle(position)** permet de préciser la position des labels par rapport aux points d'ancrage. L'argument *position* est une chaîne qui peut valoir : *"N"* pour nord, *"NE"* pour nord-est, *"NW"* pour nord-ouest, ou encore *"S"*, *"SE"*, *"SW"*. Par défaut, il vaut *center*, et dans ce cas le label est centré sur le point d'ancrage.
  - La méthode **g:Labelcolor(color)** permet de définir la couleur des labels. L'argument *color* est une chaîne représentant une couleur pour tikz. Par défaut l'argument est une chaîne vide ce qui représente la couleur courante du document.
  - La méthode **g:Labelangle(angle)** permet de préciser un angle (en degrés) de rotation des labels autour du point d'ancrage, cet angle est nul par défaut.
  - La méthode **g:Labelsize(size)** permet de gérer la taille des labels. L'argument *size* est une chaîne qui peut valoir : *"tiny"*, ou *"scriptsize"* ou *"footnotesize"*, etc. Par défaut l'argument est une chaîne vide, ce qui représente la taille *"normalsize"*.
- La méthode **g:Dlabeldot(texte,anchor,args)** permet de placer un label et de dessiner le point d'ancrage en même temps.
  - L'argument *texte* est une chaîne de caractères, c'est le label.
  - L'argument *anchor* est un complexe représentant le point d'ancrage du label.
  - L'argument *args* (facultatif) permet de définir les paramètres du label et du point, c'est une table à 4 champs :
    ```
    { pos=nil, dist=0, node_options="", mark_options="" }
    ```
    On retrouve les champs identiques à ceux de la méthode *Dlabel*, plus le champ *mark_options* qui est une chaîne de caractères qui sera passée telle quelle à l'instruction *\draw* lors du dessin du point d'ancrage.

7) Chemins : Dpath, Dspline et Dtcurve

- La fonction **path(chemin)** renvoie une ligne polygonale contenant les points constituant le *chemin*. Celui-ci est une table de complexes et d'instructions (sous forme de chaînes) par exemple :

```
{ Z(-3,2),-3,-2,"l",0,2,2,-1,"ca",3,Z(3,3),0.5,"la",1,Z(-1,5),Z(-3,2),"b" }
```

  avec :
  - *"m"* pour moveto,
  - *"l"* pour lineto,
  - *"b"* pour bézier (il faut deux points de contrôles),
  - *"s"* pour une spline cubique naturelle passant par les points cités,
  - *"c"* pour cercle (il faut un point et le centre, ou alors trois points),
  - *"ca"* pour arc de cercle (il faut 3 points, un rayon et un sens),
  - *"ea"* arc d'ellipse (il faut 3 points, un rayon rx, un rayon ry, un sens, et éventuellement une inclinaison en degrés),
  - *"e"* pour ellipse (il faut un point, le centre, un rayon rx, un rayon ry, et éventuellement une inclinaison en degrés),
  - *"cl"* pour close (ferme la composante courante),
  - *"la"* pour line arc, c'est à dire une ligne aux angles arrondis, (il faut indiquer le rayon juste avant l'instruction *"la"*),
  - *"cla"* ligne fermée aux angles arrondis (il faut indiquer le rayon juste avant l'instruction *"cla"*).
- La méthode **g:Dpath(chemin,draw_options)** fait le dessin du *chemin* (en utilisant au maximum les courbes de Bézier, y compris pour les arcs, les ellipses, etc). L'argument *draw_options* est une chaîne de caractères qui sera passée directement à l'instruction *\draw*.
  - L'argument *chemin* a été décrit ci-dessus.
  - L'argument *draw_options* est une chaîne (vide par défaut) qui sera passée telle quelle à l'instruction *\draw*.
- La fonction **spline(points,v1,v2)** renvoie sous forme de chemin (à dessiner avec Dpath) la spline cubique passant par les points de l'argument *points* (qui doit être une liste de complexes). Les arguments *v1* et *v2* sont vecteurs tangents imposés aux extrémités (contraintes), lorsque ceux-ci sont égaux à *nil*, c'est une spline cubique naturelle (c'est à dire sans contrainte) qui est calculée.
- La méthode **g:Dspline(points,v1,v2,draw_options)** fait le dessin de la spline décrite ci-dessus. L'argument *draw_options* est une chaîne de caractères qui sera passée directement à l'instruction *\draw*.

```
\begin{luadraw}{name=path_spline}
local g = graph:new{window={-5,5,-5,5},size={7,7},bg="Beige"}
local i = cpx.I
local p = {-3+2*i,-3,-2,"l",0,2,2,1,"ca",3,3+3*i,0.5,"la",
           1,-1+5*i,-3+2*i,"b",-1,"m",0,"c"}
g:Daxes( {0,1,1} )
g:Filloptions("full","blue!30",1,true)
g:Dpath(p,"line width=0.8pt")
g:Filloptions("none")
local A,B,C,D,E = -4-i,-3*i,4,3+4*i,-4+2*i
g:Lineoptions(nil,"ForestGreen",12)
g:Dspline({A,B,C,D,E},nil,-5*i) -- contrainte en E
g:Ddots({A,B,C,D,E},"fill=white,scale=1.25")
g:Show()
\end{luadraw}
```

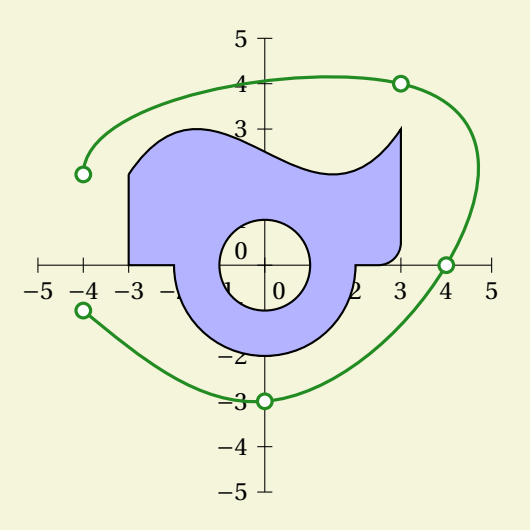

FIGURE 8 – Path et Spline

- La fonction **tcurve(L** renvoie sous forme de chemin une courbe passant par des points donnés avec des vecteurs tangents (à gauche et à droite) imposés à chaque point. *L* est une table de la forme :

```
L = {point1,{t1,a1,t2,a2}, point2,{t1,a1,t2,a2}, ..., pointN,{t1,a1,t2,a2}}
```

  point1, ..., *pointN* sont les points d'interpolation de la courbe (affixes), et chacun d'eux est suivi d'une table de la forme `{t1,a1,t2,a2}` qui précise les vecteurs tangents à la courbe à gauche du point (avec *t1* et *a1*) et à droite du point (avec *t2* et *a2*). Le vecteur tangent à gauche est donné par la formule $V_g = t_1 \times e^{ia_1\pi/180}$, donc $t1$ représente le module et $a1$ est un argument **en degrés** de ce vecteur. C'est la même chose avec *t2* et *a2* pour le vecteur tangent à droite, **mais ceux-ci sont facultatifs**, et s'ils ne sont pas précisés alors ils prennent les mêmes valeurs que *t1* et *a1*.

Deux points consécutifs seront reliés par une courbe de Bézier, la fonction calcule les points de contrôle pour avoir les vecteurs tangents souhaités.

- La méthode **g:Dtcurve(L,options)** fait le dessin du chemin obtenu par *tcurve* décrit ci-dessus. L'argument *options* est une table à deux champs :
  - *showdots=true/false* (false par défaut), cette option permet de dessiner les points d'interpolation donnés ainsi que les points de contrôles calculés, ce qui permet une visualisation des contraintes.
  - *draw_options=""*, c'est une chaîne de caractères qui sera passée directement à l'instruction *\draw*.

```
\begin{luadraw}{name=tcurve}
local g = graph:new{window={-0.5,10.5,-0.5,6.5},size={7,7,0}}
local i = cpx.I
local L = {
    1+4*i,{2,-20},
    2+3*i,{2,-70},
    4+i/2,{3,0},
    6+3*i,{4,15},
    8+6*i,{4,0,4,-90}, -- point anguleux
    10+i,{3,-15}}
g:Dgrid({0,10+6*i},{gridstyle="dashed"})
g:Daxes(nil,{limits={{0,10},{0,6}},originpos={"center","center"},
 ↪  arrows="->"})
g:Dtcurve(L,{showdots=true,draw_options="line width=0.8pt,red"})
g:Show()
\end{luadraw}
```

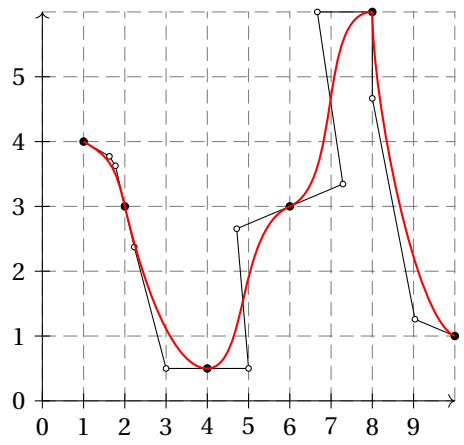

FIGURE 9 – Courbe d'interpolation avec vecteurs tangents imposés

8) Axes et grilles

Varaibles globales utilisées pour les axes et les grilles :

- *maxGrad = 100* : nombre max de graduations sur un axe.
- *defaultlabelshift = 0.125* : lorsqu'une grille est dessinée avec les axes (option *grid=true*) les labels sont automatiquement décalés le long de l'axe avec cette variable.
- *defaultxylabelsep = 0* : définit la distance par défaut entre les labels et les graduations.
- *defaultlegendsep = 0.2* : définit la distance par défaut entre la légende et l'axe.
- *dollar = true* : pour ajouter des dollars autour des labels des graduations.

8.1 Daxes

Le tracé des axes s'obtient avec la méthode **g:Daxes({A,xpas,ypas}, options)**.

- Le premier argument précise le point d'intersection des deux axes (c'est le complexe A), le pas des graduations sur l'axe Ox (c'est *xpas*) et le pas des graduations sur Oy (c'est *ypas*). Par défaut le point A est l'origine Z$(0,0)$, et les deux pas sont égaux à 1.
- L'argument *options* est une table précisant les options possibles. Voici ces options avec leur valeur par défaut :
  - `showaxe={1,1}`. Cette option précise si les axes doivent être tracés ou pas (1 ou 0). La première valeur est pour l'axe Ox et la seconde pour l'axe Oy.
  - `arrows="-"`. Cette option permet d'ajouter ou non une flèche aux axes (pas de flèche par défaut, mettre "->" pour ajouter une flèche).
  - `limits={"auto","auto"}`. Cette option permet de préciser l'étendue des deux axes (première valeur pour Ox, seconde valeur pour Oy). La valeur "auto" signifie que c'est la droite en entier, mais on peut préciser les abscisses extrêmes, par exemple : `limits={{-4,4},"auto"}`.
  - `gradlimits={"auto","auto"}`. Cette option permet de préciser l'étendue des graduations sur les deux axes (première valeur pour Ox, seconde valeur pour Oy). La valeur "auto" signifie que c'est la droite en entier, mais on peut préciser les graduations extrêmes, par exemple : `gradlimits={{-4,4},{-2,3}}`.

- `unit={"",""}`. Cette option permet de préciser de combien en combien vont les graduations sur les axes. La valeur par défaut ("") signifie qu'il faut prendre la valeur du pas (*xpas* sur Ox, ou *ypas* sur Oy), SAUF lorsque l'option `labeltext` n'est pas la chaîne vide, dans ce cas *unit* prend la valeur 1.
- `nbsubdiv={0,0}`. Cette option permet de préciser le nombre de subdivisions entre deux graduations principales sur l'axe.
- `tickpos={0.5,0.5}`. Cette option précise la position des graduations par rapport à chaque axe, ce sont deux nombres entre 0 et 1, la valeur par défaut de 0.5 signifie qu'ils sont centrés sur l'axe. (0 et 1 représentent les extrémités).
- `tickdir={"auto","auto"}`. Cette option indique la direction des graduations sur l'axe. Cette direction est un vecteur (complexe) non nul. La valeur par défaut "auto" signifie que les graduations sont orthogonales à l'axe.
- `xyticks={0.2,0.2}`. Cette option précise la longueur des graduations sur l'axe.
- `xylabelsep={0,0}`. Cette option précise la distance entre les labels et les graduations sur l'axe.
- `originpos={"right","top"}`. Cette option précise la position du label à l'origine sur l'axe, les valeurs possibles sont : "none", "center", "left", "right" pour Ox, et "none", "center", "bottom", "top" pour Oy.
- `originnum={A.re,A.im}`. Cette option précise la valeur de la graduation au croisement des axes (graduation numéro 0).
  La formule qui définit le label à la graduation numéro n est : **(originnum + unit*n)"labeltext"/labelden**.
- `originloc=A`. Cette option précise le point de croisement des axes.
- `legend={"",""}`. Cette option permet de préciser une légende pour l'axe.
- `legendpos={0.975,0.975}`. Cette option précise la position (entre 0 et 1) de la légende par rapport à chaque axe.
- `legendsep={0.2,0.2}` . Cette option précise la distance entre la légende et l'axe. La légende est de l'autre côté de l'axe par rapport aux graduations.
- `legendangle={"auto","auto"}`. Cette option précise l'angle (en degrés) que doit faire la légende pour l'axe. La valeur "auto" par défaut signifie que la légende doit être parallèle à l'axe si l'option *labelstyle* est aussi à "auto", sinon la légende est horizontale.
- `labelpos={"bottom","left"}`.Cette option précise la position des labels par rapport à l'axe. Pour l'axe Ox, les valeurs possibles sont : "none", "bottom" ou "top", pour l'axe Oy c'est : "none", "right" ou "left".
- `labelden={1,1}`. Cette option précise le dénominateur des labels (entier) pour l'axe. La formule qui définit le label à la graduation numéro n est : **(originnum + unit*n)"labeltext"/labelden**.
- `labeltext={"",""}`. Cette option définit le texte qui sera ajouté au numérateur des labels pour l'axe.
- `labelstyle={"S","W"}`. Cette option définit le style des labels pour chaque axe. Les valeurs possibles sont "auto","N", "NW", "W", "SW", "S", "SE", "E".
- `labelangle={0,0}`. Cette option définit pour chaque axe l'angle des labels en degrés par rapport à l'horizontale.
- `labelcolor={"",""}`. Cette option permet de choisir une couleur pour les labels sur chaque axe. La chaîne vide représente la couleur par défaut.
- `labelshift={0,0}`. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe).
- `nbdeci={2,2}`. Cette option précise le nombre de décimales pour les valeurs numériques sur l'axe.
- `numericFormat={0,0}`. Cette option précise le type d'affiche numérique (non encore implémenté).
- `myxlabels=""`. Cette option permet d'imposer des labels personnels sur l'axe Ox. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1", pos2,"text2",...}`. Le nombre *pos1* représente une abscisse dans le repère (A,xpas), ce qui correspond au point d'affixe A+pos1$*$xpas.
- `myylabels=""`. Cette option permet d'imposer des labels personnels sur l'axe Oy. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1", pos2,"text2",...}`. Le nombre *pos1* représente une abscisse dans le repère (A,i*ypas), ce qui correspond au point d'affixe A+pos1$*$ypas$*i$.
- `grid=false`. Cette option permet d'ajouter ou non une grille.
- `drawbox=false`. Cette option de dessiner les axes sous la forme d'une boite, dans ce cas, les graduations sont sur le côté gauche et le côté bas.
- `gridstyle="solid"`. Cette option définit le style ligne pour la grille principale.
- `subgridstyle="solid"`. Cette option définit le style ligne pour la grille secondaire. Une grille secondaire apparaît lorsqu'il y a des subdivisions sur un des axes.

– gridcolor="gray". Ceci définit la couleur de la grille principale.
– subgridcolor="lightgray". Ceci définit la couleur de la grille secondaire.
– gridwidth=4. Épaisseur de trait de la grille principale (ce qui fait 0.4pt).
– subgridwidth=2. Épaisseur de trait de la grille secondaire (ce qui fait 0.2pt).

```
\begin{luadraw}{name=axes_grid}
local g = graph:new{window={-6.5,6.5,-3.5,3.5}, size={7,7,0}}
local i, pi, a = cpx.I, math.pi, math.sqrt(2)
local f = function(x) return 2*a*math.sin(x) end
g:Labelsize("footnotesize"); g:Linewidth(8)
g:Daxes({0,pi/2,a},{labeltext={"\\pi","\\sqrt{2}"},
        labelden={2,1},nbsubdiv={1,1},grid=true,arrows="->"})
g:Lineoptions("solid","Crimson",12)
g:Dcartesian(f, {x={-2*pi,2*pi}})
g:Show()
\end{luadraw}
```

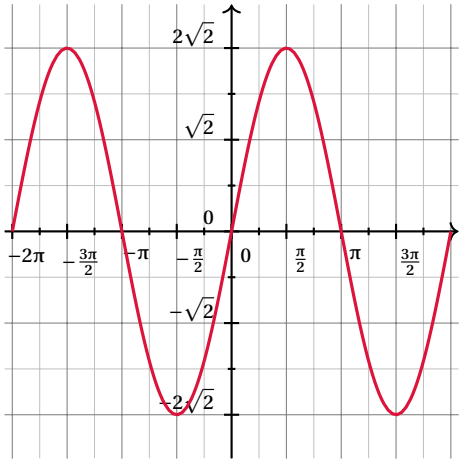

FIGURE 10 – Exemple avec axes avec grille

8.2 DaxeX et DaxeY

Les méthodes **g:DaxeX({A,xpas}, options)** et **g:DaxeY({A,ypas}, options)** permettent de tracer les axes séparément.

- Le premier argument précise le point servant d'origine (c'est le complexe A) et le pas des graduations sur l'axe. Par défaut le point A est l'origine Z(0,0), et le pas est égal à 1.
- L'argument *options* est une table précisant les options possibles. Voici ces options avec leur valeur par défaut :
  – showaxe=1. Cette option précise si l'axe doit être tracé ou non (1 ou 0).
  – arrows="-". Cette option permet d'ajouter ou non une flèche à l'axe (pas de flèche par défaut, mettre "->" pour ajouter une flèche).
  – limits="auto". Cette option permet de préciser l'étendue des deux axes. La valeur "auto" signifie que c'est la droite en entier, mais on peut préciser les abscisses extrêmes, par exemple : limits={-4,4}.
  – gradlimits="auto". Cette option permet de préciser l'étendue des graduations sur les deux axes. La valeur "auto" signifie que c'est la droite en entier, mais on peut préciser les graduations extrêmes, par exemple : gradlimits={-2,3}.
  – unit="". Cette option permet de préciser de combien en combien vont les graduations sur l'axe. La valeur par défaut ("") signifie qu'il faut prendre la valeur du pas, SAUF lorsque l'option labeltext n'est pas la chaîne vide, dans ce cas *unit* prend la valeur 1.
  – nbsubdiv=0. Cette option permet de préciser le nombre de subdivisions entre deux graduations principales.
  – tickpos=0.5. Cette option précise la position des graduations par rapport à l'axe, ce sont deux nombres entre 0 et 1, la valeur par défaut de 0.5 signifie qu'ils sont centrés sur l'axe. (0 et 1 représentent les extrémités).
  – tickdir="auto". Cette option indique la direction des graduations sur l'axe. Cette direction est un vecteur (complexe) non nul. La valeur par défaut "auto" signifie que les graduations sont orthogonales à l'axe.
  – xyticks=0.2. Cette option précise la longueur des graduations.
  – xylabelsep=0. Cette option précise la distance entre les labels et les graduations.
  – originpos="center". Cette option précise la position du label à l'origine sur l'axe, les valeurs possibles sont : "none", "center", "left", "right" pour Ox, et "none", "center", "bottom", "top" pour Oy.
  – originnum=A.re pour Ox et originnum=A.im pour Oy. Cette option précise la valeur de la graduation à l'origine (graduation numéro 0).
  La formule qui définit le label à la graduation numéro n est : **(originnum + unit*n)"labeltext"/labelden**.
  – legend="". Cette option permet de préciser une légende pour l'axe.
  – legendpos=0.975. Cette option précise la position (entre 0 et 1) de la légende par rapport à l'axe.

- `legendsep=0.2`. Cette option précise la distance entre la légende et l'axe. La légende est de l'autre côté de l'axe par rapport aux graduations.
- `legendangle="auto"`. Cette option précise l'angle (en degrés) que doit faire la légende pour l'axe. La valeur "auto" par défaut signifie que la légende doit être parallèle à l'axe si l'option *labelstyle* est aussi à "auto", sinon la légende est horizontale.
- `labelpos="bottom"` pour Ox et `labelpos="left"` pour Oy. Cette option précise la position des labels par rapport à l'axe. Pour l'axe Ox, les valeurs possibles sont : "none", "bottom" ou "top", pour l'axe Oy c'est : "none", "right" ou "left".
- `labelden=1`. Cette option précise le dénominateur des labels (entier) pour l'axe. La formule qui définit le label à la graduation numéro n est : **(originnum + unit*n)"labeltext"/labelden**.
- `labeltext=""`. Cette option définit le texte qui sera ajouté au numérateur des labels.
- `labelstyle="S"` pour Ox et `labelstyle="W"` pour Oy. Cette option définit le style des labels. Les valeurs possibles sont "auto","N", "NW", "W", "SW", "S", "SE", "E".
- `labelangle=0`. Cette option définit l'angle des labels en degrés par rapport à l'horizontale.
- `labelcolor=""`. Cette option permet de choisir une couleur pour les labels. La chaîne vide représente la couleur courante du texte.
- `labelshift=0`. Cette option permet de définir un décalage systématique pour les labels sur l'axe (décalage de long de l'axe).
- `nbdeci=2`. Cette option précise le nombre de décimales pour les labels numériques.
- `numericFormat=0`. Cette option précise le type d'affiche numérique (non encore implémenté).
- `mylabels=""`. Cette option permet d'imposer des labels personnels. Lorsqu'il y en a, la valeur passée à l'option doit être une liste du type : `{pos1,"text1", pos2,"text2",...}`. Le nombre *pos1* représente une abscisse dans le repère (A,xpas) pour Ox, ou (A,ypas∗i) pour Oy, ce qui correspond au point d'affixe A+pos1∗xpas pour Ox, et A+pos1∗ypas∗i pour Oy.

8.3 Dgrid

La méthode **g:Dgrid({A,B},options** permet le dessin d'une grille.

- Le premier argument est obligatoire, il précise le coin inférieur gauche (c'est le complexe A), le coin supérieur droit (c'est le complexe B) de la grille.
- L'argument *options* est une table précisant les options possibles. Voici ces options avec leur valeur par défaut :
  - `unit={1,1}`. Cette option définit les unités sur les axes pour la grille principale.
  - `gridwidth=4`. Cette option définit l'épaisseur du trait de la grille principale (0.4pt par défaut).
  - `gridcolor="gray"`. Couleur grille de la grille principale.
  - `gridstyle="solid"`. Style de trait pour la grille principale.
  - `nbsubdiv={0,0}`. Nombre de subdivisions (pour chaque axe) entre deux traits de la grille principale. Ces subdivisions déterminent la grille secondaire.
  - `subgridcolor="lightgray"`. Couleur de la grille secondaire.
  - `subgridwidth=2`. Épaisseur du trait de la grille secondaire (0.2pt par défaut).
  - `subgridstyle="solid"`. Style de trait pour la grille secondaire.
  - `originloc=A`. Localisation de l'origine de la grille.

Exemple : il est possible de travailler dans un repère non orthogonal. Voici un exemple où l'axe Ox est conservé, mais la première bissectrice devient le nouvel axe Oy, on modifie pour cela la matrice de transformation du graphe. À partir de cette modification les affixes représentent les coordonnées dans le nouveau repère.

```
\begin{luadraw}{name=axes_non_ortho}
local g = graph:new{window={-5.25,5.25,-4,4},size={7,7}}
local i, pi = cpx.I, math.pi
local f = function(x) return 2*math.sin(x) end
g:Setmatrix({0,1,1+i}); g:Labelsize("small")
g:Dgrid({-5-4*i,5+4*i},{gridstyle="dashed"})
g:Daxes({0,1,1}, {arrows="-Stealth"})
g:Lineoptions("solid","ForestGreen",12)
g:Dcartesian(f,{x={-5,5}})
g:Dcircle(0,3,"Crimson")
g:DlineEq(1,0,3,"Navy") -- droite d'équation x=-3
g:Lineoptions("solid","black",8)
g:DtangentC(f,pi/2,1.5,"<->")
g:Dpolyline({pi/2,pi/2+2*i,2*i},"dotted")
g:Ddots(Z(pi/2,2))
g:Dlabeldot("$\\frac{\\pi}2$",pi/2,{pos="SW"})
g:Show()
\end{luadraw}
```

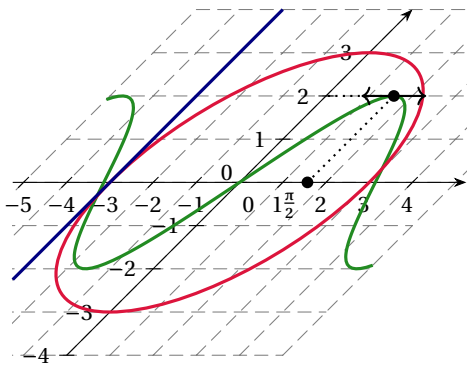

FIGURE 11 – Exemple de repère non orthogonal

8.4 Dgradbox

La méthode **g:Dgradbox({A,B,xpas,ypas},options** permet le dessin d'une boîte graduée.

- Le premier argument est obligatoire, il précise le coin inférieur gauche (c'est le complexe A) et le coin supérieur droit (c'est le complexe B) de la boîte, ainsi que le pas sur chaque axe.
- L'argument *options* est une table précisant les options possibles. Ce sont les mêmes que pour les axes, mises à part certaines valeurs par défaut. À celles-ci s'ajoute l'option suivante : `title=""` qui permet d'ajouter un titre en haut de la boite, attention cependant à laisser suffisamment de place pour cela.

```
\begin{luadraw}{name=gradbox}
local g = graph:new{window={-5,4,-5.5,5},size={7,7}}
local i, pi = cpx.I, math.pi
local h = function(x) return x^2/2-2 end
local f = function(x) return math.sin(3*x)+h(x) end
g:Dgradbox({-pi-4*i,pi+4*i,pi/3,1},{grid=true,originloc=0,
  originnum={0,0},labeltext={"\\pi",""},labelden={3,1},
  title="\\textbf{Title}",legend={"Legend $x$","Legend $y$"}})
g:Saveattr(); g:Viewport(-pi,pi,-4,4) -- on limite la vue (clip)
g:Filloptions("full","blue",0.6); g:Linestyle("noline")
g:Ddomain2(f,h,{x={-pi/2,2*pi/3}})
g:Filloptions("none",nil,1); g:Lineoptions("solid",nil,8)
g:Dcartesian(h,{x={-pi,pi}, draw_options="DarkBlue"})
g:Dcartesian(f,{x={-pi,pi},draw_options="Crimson"})
g:Restoreattr()
g:Show()
\end{luadraw}
```

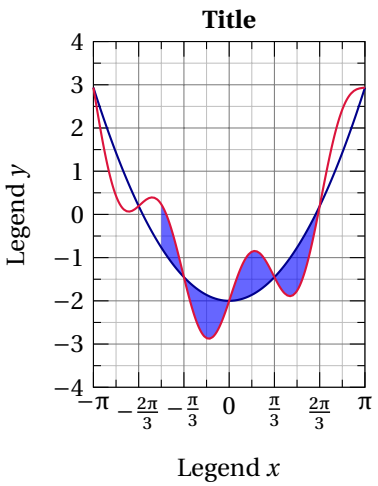

FIGURE 12 – Utilisation de Dgradbox

9) Calculs sur les couleurs

Dans l'environnement *luadraw* les couleurs sont des chaînes de caractères qui doivent correspondre à des couleurs connues de tikz. Le package *xcolor* est fortement conseillé pour ne pas être limité aux couleurs de bases.

Afin de pouvoir faire des manipulations sur les couleurs, celles-ci ont été définies (dans le module *luadraw_colors.lua*) sous la forme de tables de trois composantes : rouge, vert, bleu, chaque composante étant un nombre entre 0 et 1, et avec leur nom au format *svgnames* du package *xcolor*, par exemple on y trouve (entre autres) les déclarations :

```
AliceBlue = {0.9412, 0.9725, 1}
AntiqueWhite = {0.9804, 0.9216, 0.8431}
Aqua = {0.0, 1.0, 1.0}
Aquamarine = {0.498, 1.0, 0.8314}
```

On pourra se référer à la documentation de *xcolor* pour avoir la liste de ces couleurs.

Pour utiliser celles-ci dans l'environnement *luadraw*, on peut :

- soit les utiliser avec leur nom si on a déclaré dans le préambule : `\usepackage[svgnames]{xcolor}`, par exemple : `g:Linecolor("AliceBlue")`,
- soit les utiliser avec la fonction **rgb()** de *luadraw*, par exemple : `g:Linecolor(rgb(AliceBlue))`. Par contre, avec cette fonction *rgb()*, pour changer localement de couleur il faut faire comme ceci (exemple) : `g:Dpolyline(L,"color="..rgb(AliceBlue))`, ou `g:Dpolyline(L,"fill="..rgb(AliceBlue))`. Car la fonction *rgb()* ne renvoie pas un nom de couleur, mais une définition de couleur.

Fonctions pour la gestion des couleurs :

- La fonction **rgb(r,g,b)** ou **rgb({r,g,b})**, renvoie la couleur sous forme d'une chaîne de caractères compréhensible par tikz dans les options `color=...` et `fill=...`. Les valeurs de *r*, *g* et *b* doivent être entre 0 et 1.
- La fonction **hsb(h,s,b)** renvoie la couleur sous forme d'une chaîne de caractères compréhensible par tikz. L'argument *h* (hue) doit être un nombre enter 0 et 360, l'argument *s* (saturation) doit être entre 0 et 1, et l'argument *b* (brightness) doit être aussi entre 0 et 1.
- La fonction **mixcolor(color1,proportion1 color2,proportion1,...,colorN,proportionN)** mélange les couleurs *color1*, ...,*colorN* dans les proprtions demandées et renvoie la couleur qui en résulte sous forme d'une chaîne de caractères compréhensible par tikz. Chacune des couleurs doit être une table de trois composantes `{r,g,b}`.
- La fonction **palette(colors,pos)** : l'argument *colors* est une liste (table) de couleurs au format `{r,b,g}`, l'argument *pos* est un nombre entre 0 et 1, la valeur 0 correspond à la première couleur de la liste et la valeur 1 à la dernière. La fonction calcule et renvoie (sous forme de chaîne) la couleur correspondant à la position *pos* dans la liste par interpolation linéaire.

On peut également utiliser toutes les possibilités habituelles de tikz pour la gestion des couleurs.

```
\begin{luadraw}{name=palette}
local g = graph:new{window={-5,5,-1,1},size={7,7},
      margin={0.1,0.1,0.1,0.1},border=true}
local i = cpx.I
local colors = {Purple,Indigo,Blue,Green,Yellow,Orange,Red}
local N = 200
g:Linewidth(18)
for k = 1, N do
    local pos = (k-1)/(N-1)
    local x = -5+10*pos
    g:Dpolyline({x-i,x+i},"color="..palette(colors,pos))
end
g:Show()
\end{luadraw}
```

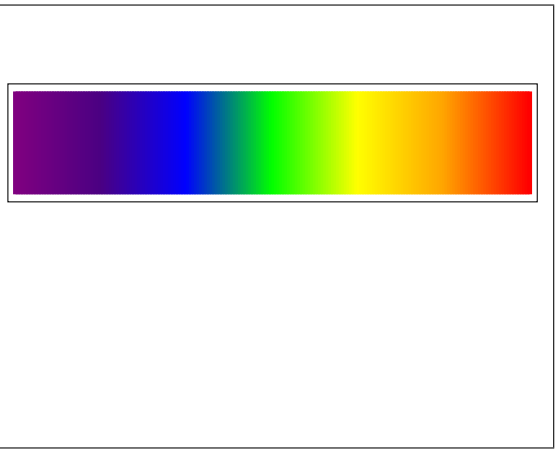

FIGURE 13 – Utilisation de la fonction palette()

III Calculs sur les listes

1) concat

La fonction **concat{table1, table2, ... }** concatène toutes les tables passées en argument, et renvoie la table qui en résulte.

- Chaque argument peut être un réel un complexe ou une table.
- Exemple : l'instruction `concat( 1,2,3,{4,5,6},7 )` renvoie la table *{1,2,3,4,5,6,7}*.

2) cut

La fonction **cut(L,A,before)** permet de couper la ligne polygonale *L* au point *A* qui est sensé être situé sur la ligne *L*. Si l'argument *before* vaut *true*, c'est la partie située avant *A* qui sera coupée, et la partie située après *A* qui sera renvoyée, sinon c'est l'inverse (*before* vaut *false* par défaut). Le résultat est une ligne polygonale (liste de listes de complexes).

3) getbounds

- La fonction **getbounds(L)** renvoie les bornes xmin,xmax,ymin,ymax de la ligne polygonale *L*.
- Exemple : `local xmin, xmax, ymin, ymax = getbounds(L)` (où *L* désigne une ligne polygonale).

4) getdot

La fonction **getdot(x,L)** renvoie le point d'abscisse *x* (réel entre 0 et 1) le long de la composante connexe *L* (liste de complexes). L'abscisse 0 correspond au premier point et l'abscisse 1 au dernier, plus généralement, *x* correspond à un pourcentage de la longueur de *L*.

5) insert

La fonction **insert(table1, table2, pos)** insère les éléments de *table2* dans *table1* à la position *pos*.

- L'argument *table2* peut être un réel, un complexe ou une table.
- L'argument *table1* doit être une variable qui désigne une table, celle-ci sera modifiée par la fonction.
- Si l'argument *pos* vaut *nil*, l'insertion se fait à la fin de *table1*.
- Exemple : si une variable *L* vaut *{1,2,6}*, alors après l'instruction `insert(L, {3,4,5},3)`, la variable *L* sera égale à *{1,2,3,4,5,6}*.

6) interD

La fonction **interD(d1,d2)** renvoie le point d'intersection des droites *d1* et *d2*, une droite est une liste de deux complexes : un point de la droite et un vecteur directeur.

7) interDL

La fonction **interDL(d,L)** renvoie la liste des points d'intersection entre la droite *d* et la ligne polygonale *L*.

8) interL

La fonction **interL(L1,L2)** renvoie la liste des points d'intersection des lignes polygonales définies par *L1* et *L2*, ces deux arguments sont deux listes de complexes ou deux listes de listes de complexes).

9) interP

La fonction **interP(P1,P2)** renvoie la liste des points d'intersection des chemins définis par *P1* et *P2*, ces deux arguments sont deux listes de complexes et d'instructions (voir *Dpath*).

10) linspace

La fonction **linspace(a,b,nbdots)** renvoie une liste de *nbdots* nombres équirépartis de *a* jusqu'à *b*. Par défaut *nbdots* vaut 50.

11) map

La fonction **map(f,list)** applique la fonction *f* à chaque élément de la *list* et renvoie la table des résultats. Lorsqu'un résultat vaut *nil*, c'est le complexe *cpx.Jump* qui est inséré dans la liste.

12) merge

La fonction **merge(L)** recolle si c'est possible, les composantes connexes de *L* qui doit être une liste de listes de complexes, la fonction renvoie le résultat.

13) range

La fonction **range(a,b,step)** renvoie la liste des nombres de *a* jusqu'à *b* avec un pas égal à *step*, celui-ci vaut 1 par défaut.

14) Fonctions de clipping

- La fonction **clipseg(A,B,xmin,xmax,ymin,ymax)** clippe le segment *[A,B]* avec la fenêtre *[xmin,xmax]x[ymin,ymax]*et renvoie le résultat.
- La fonction **clipline(d,xmin,xmax,ymin,ymax)** clippe la droite *d* avec la fenêtre *[xmin,xmax]x[ymin,ymax]* et renvoie le résultat. La droite *d* est une liste de deux complexes : un point et un vecteur directeur.
- La fonction **clippolyline(L,xmin,xmax,ymin,ymax,close)** clippe ligne polygonale *L* avec *[xmin,xmax]x[ymin,ymax]* et renvoie le résultat. L'argument *L* est une liste de complexes ou une liste de listes de complexes. L'argument facultatif *close* (false par défaut) indique si la ligne polygonale doit être refermée.
- La fonction **clipdots(L,xmin,xmax,ymin,ymax)** clippe la liste de points *L* avec la fenêtre *[xmin,xmax]x[ymin,ymax]* et renvoie le résultat (les points extérieurs sont simplement exclus). L'argument *L* est une liste de complexes ou une liste de listes de complexes.

15) Ajout de fonctions mathématiques

Outre les fonctions associées aux méthodes graphiques qui font des calculs et renvoient une ligne polygonale (comme *cartesian, periodic, implicit, odesolve,* etc), le paquet *luadraw* ajoute quelques fonctions mathématiques qui ne sont pas proposées nativement dans le module *math*.

15.1 int

La fonction **int(f,a,b)** renvoie une valeur approchée de l'intégrale de la fonction f sur l'intervalle $[a;b]$. La fonction f est à variable réelle et à valeurs réelles ou complexes. La méthode utilisée est la méthode de Simpson accélérée deux fois avec la méthode Romberg.

Exemple :

```
$\int_0^1 e^{t^2}\mathrm d t \approx \directlua{tex.sprint(int(function(t) return math.exp(t^2) end, 0, 1))}$
```

Résultat : $\int_0^1 e^{t^2}\,\mathrm{d}t \approx 1.4626517459589$.

15.2 gcd

La fonction **gcd(a,b)** renvoie le plus grand diviseur commun entre *a* et *b*.

15.3 lcm

La fonction **lcm(a,b)** renvoie le plus petit diviseur commun strictement positif entre *a* et *b*.

15.4 solve

La fonction **solve(f,a,b,n)** fait une résolution numérique de l'équation $f(x) = 0$ dans l'intervalle $[a;b]$, celui-ci est subdivisé en *n* morceaux (*n* vaut 25 par défaut). La fonction renvoie une liste de résultats ou bien *nil*. La méthode utilisée est une variante de Newton.

Exemple 1 :

```
\begin{luacode}
resol = function(f,a,b)
    local y = solve(f,a,b)
    if y == nil then tex.sprint("\\emptyset")
    else
        local str = y[1]
        for k = 2, #y do
```

```
            str = str.., ".. y[k]
        end
        tex.sprint(str)
    end
end
\end{luacode}
\def\solve#1#2#3{\directlua{resol(#1,#2,#3)}}%
\begin{luacode}
f1 = function(x) return math.cos(x)-x end
f2 = function(x) return x^3-2*x^2+1/2 end
\end{luacode}
La résolution de l'équation $\cos(x)=x$ dans $[0;\frac{\pi}2]$ donne $\solve{f1}{0}{math.pi/2}$.\par
La résolution de l'équation $\cos(x)=x$ dans $[\frac{\pi}2;\pi]$ donne $\solve{f1}{math.pi/2}{math.pi}$.\par
La résolution de l'équation $x^3-2x^2+\frac12=0$ dans $[-1;2]$ donne : $\{\solve{f2}{-1}{2}\}$.
```

Résultat :

La résolution de l'équation $\cos(x) = x$ dans $[0; \frac{\pi}{2}]$ donne 0.73908513321516.
La résolution de l'équation $\cos(x) = x$ dans $[\frac{\pi}{2}; \pi]$ donne $\emptyset$.
La résolution de l'équation $x^3-2x^2+\frac{1}{2} = 0$ dans $[-1;2]$ donne : $\{-0.45160596295578, 0.59696828323732, 1.8546376797185\}$.

Exemple 2 : on souhaite tracer la courbe de la fonction f définie par la condition :

$$\forall x \in \mathbf{R},\ \int_x^{f(x)} \exp(t^2)\mathrm{d}t = 1.$$

On a deux méthodes possibles :

1. On considère la fonction G: $(x,y) \mapsto \int_x^y \exp(t^2)\mathrm{d}t - 1$, et on dessine la courbe implicite d'équation $\mathrm{G}(x,y) = 0$.
2. On détermine un réel y_0 tel que $\int_0^{y_0} \exp(t^2)\mathrm{d}t = 1$ et on dessine la solution de l'équation différentielle $y' = e^{x^2-y^2}$ vérifiant la condition initiale $y(0) = y_0$.

Dessinons les deux :

```
\begin{luadraw}{name=int_solve}
local g = graph:new{window={-3,3,-3,3},size={7,7}}
local h = function(t) return math.exp(t^2) end
local G = function(x,y) return int(h,x,y)-1 end
local H = function(y) return G(0,y) end
local F = function(x,y) return math.exp(x^2-y^2) end
local y0 = solve(H,0,1)[1] -- solution de H(x)=0
g:Daxes({0,1,1}, {arrows="->"})
g:Dimplicit(G, {draw_options="line width=4.8pt,Pink"})
g:Dodesolve(F,0,y0,{draw_options="line width=0.8pt"})
g:Lineoptions("dashed","gray",4)
g:DlineEq(1,-1,0); g:DlineEq(1,1,0) -- bissectrices
g:Dlabel("${\\mathcal C}_f$",Z(2.15,2),{pos="S"})
g:Show()
\end{luadraw}
```

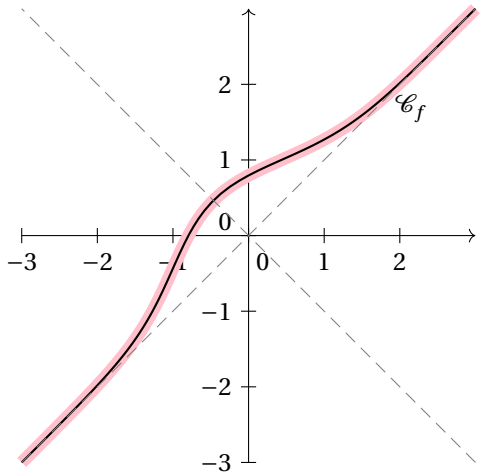

FIGURE 14 – Fonction f définie par $\int_x^{f(x)} \exp(t^2)\mathrm{d}t = 1$.

On voit que les deux courbes se superposent bien, cependant la première méthode (courbe implicite) est beaucoup plus gourmande en calculs, la méthode 2 est donc préférable.

IV Transformations

Dans ce qui suit :

- l'argument L est soit un complexe, soit une liste de complexes soit une liste de listes de complexes,
- la droite d est une liste de deux complexes : un point de la droite et un vecteur directeur.

1) affin

La fonction **affin(L,d,v,k)** renvoie l'image de *L* par l'affinité de base la droite *d*, parallèlement au vecteur *v* et de rapport *k*.

2) ftransform

La fonction **ftransform(L,f)** renvoie l'image de *L* par la fonction *f* qui doit être une fonction de la variable complexe. Si un des éléments de *L* est le complexe *cpx.Jump* alors celui-ci est renvoyé tel quel dans le résultat.

3) hom

La fonction **hom(L,factor,center)** renvoie l'image de *L* par l'homothétie de centre *center* et de rapport *factor*. Par défaut, l'argument *center* vaut 0.

4) inv

La fonction **inv(L, center, r)** renvoie l'image de *L* par l'inversion par rapport au cercle de centre *center* et de rayon *r*.

5) proj

La fonction **proj(L,d)** renvoie l'image de *L* par la projection orthogonale sur la droite *d*.

6) projO

La fonction **projO(L,d,v)** renvoie l'image de *L* par la projection sur la droite *d* parallèlement au vecteur *v*.

7) rotate

La fonction **rotate(L,angle,center)** renvoie l'image de *L* par la rotation de centre *center* et d'angle *angle* (en degrés). Par défaut, l'argument *center* vaut 0.

8) shift

La fonction **shift(L,u)** renvoie l'image de *L* par la translation de vecteur *u*.

9) simil

La fonction **simil(L,factor,angle,center)** renvoie l'image de *L* par la similitude de centre *center*, de rapport *factor* et d'angle *angle* (en degrés). Par défaut, l'argument *center* vaut 0.

10) sym

La fonction **sym(L,d)** renvoie l'image de *L* par la symétrie orthogonale d'axe la droite *d*.

11) symG

La fonction **symG(L,d,v)** renvoie l'image de *L* par la symétrie par rapport à la droite *d* suivie de la translation de vecteur *v* (symétrie glissée).

12) symO

La fonction **symO(L,d)** renvoie l'image de *L* par la symétrie par rapport à la droite *d* et parallèlement au vecteur *v* (symétrie oblique).

```
\begin{luadraw}{name=Sierpinski}
local g = graph:new{window={-5,5,-5,5},size={7,7}}
local i = cpx.I
local rand = math.random
local A, B, C = 5*i, -5-5*i, 5-5*i -- triangle initial
local T, niv = {{A,B,C}}, 5
for k = 1, niv do
    T = concat( hom(T,0.5,A), hom(T,0.5,B), hom(T,0.5,C) )
end
for _,cp in ipairs(T) do
    g:Filloptions("full", rgb(rand(),rand(),rand()))
    g:Dpolyline(cp,true)
end
g:Show()
\end{luadraw}
```

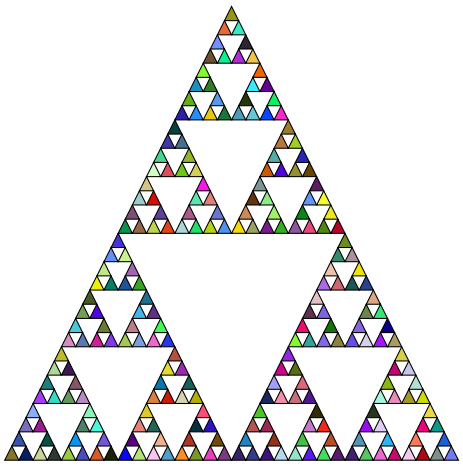

FIGURE 15 – Utilisation de transformations

V Calcul matriciel

Si f est une application affine du plan complexe, on appellera matrice de f la liste (table) :

```
1 { f(0), Lf(1), Lf(i) }
```

où Lf désigne la partie linéaire de f (on a L$f(1) = f(1) - f(0)$ et L$f(i) = f(i) - f(0)$). La matrice identité est notée *ID* dans le paquet *luadraw*, elle correspond simplement à la liste `{0,1,i}`.

1) Calculs sur les matrices

1.1 applymatrix et applyLmatrix

- La fonction **applymatrix(z,M)** applique la matrice M au complexe z et renvoie le résultat (ce qui revient à calculer $f(z)$ si M est la matrice de f). Lorsque z est le complexe *cpx.Jump* alors le résultat est *cpx.Jump*. Lorsque z est une chaîne de caractères alors la fonction renvoie z.
- La fonction **applyLmatrix(z,M)** applique la partie linéaire la matrice M au complexe z et renvoie le résultat (ce qui revient à calculer L$f(z)$ si M est la matrice de f). Lorsque z est le complexe *cpx.Jump* alors le résultat est *cpx.Jump*.

1.2 composematrix

La fonction **composematrix(M1,M2)** effectue le produit matriciel M1 × M2 et renvoie le résultat.

1.3 invmatrix

La fonction **invmatrix(M)** calcule et renvoie l'inverse de la matrice M lorsque cela est possible.

1.4 matrixof

- La fonction **matrixof(f)** calcule et renvoie la matrice de f (qui doit être une application affine du plan complexe.
- Exemple : `matrixof( function(z) return proj(z,{0,Z(1,-1)}) end )` renvoie `{0,Z(0.5,-0.5),Z(-0.5,0.5)}` (matrice de la projection orthogonale sur la deuxième bissectrice).

1.5 mtransform et mLtransform

- La fonction **mtransform(L,M)** applique la matrice M à la liste L et renvoie le résultat. L doit être une liste de complexes ou une liste de listes de complexes, si l'un d'eux est le complexe *cpx.Jump* ou une chaîne de caractères alors il est inchangé (donc renvoyé tel quel).

- La fonction **mLtransform(L,M)** applique la partie linéaire la matrice M à la liste L et renvoie le résultat. L doit être une liste de complexes, si l'un d'eux est le complexe *cpx.Jump* alors il est inchangé.

2) Matrice associée au graphe

Lorsque l'on crée un graphe dans l'environnement *luadraw*, par exemple :

```
local g = graph:new{window={-5,5,-5,5},size={7,7}}
```

l'objet *g* créé possède une matrice de transformation qui est initialement l'identité. Toutes les méthodes graphiques utilisées appliquent automatiquement la matrice de transformation du graphe. Cette matrice est désignée par `g.matrix`, mais pour manipuler celle-ci, on dispose des méthodes qui suivent.

2.1 g:IDmatrix()

La méthode **g:IDmatrix()** réaffecte l'identité à la matrice du graphe *g*.

2.2 g:Composematrix()

La méthode **g:Composematrix(M)** multiplie la matrice du graphe *g* par la matrice *M* (avec *M* à droite) et le résultat est affecté à la matrice du graphe. L'argument *M* doit donc être une matrice.

2.3 g:Mtransform()

La méthode **g:Mtransform(L)** applique la matrice du graphe *g* à *L* et renvoie le résultat, l'argument *L* doit être une liste de complexes, ou une liste de listes de complexes.

2.4 g:MLtransform()

La méthode **g:MLtransform(L)** applique la partie linéaire de la matrice du graphe *g* à *L* et renvoie le résultat, l'argument *L* doit être une liste de complexes, ou une liste de listes de complexes.

```
\begin{luadraw}{name=Pythagore}
local g = graph:new{window={-15,15,0,22},size={7,7}}
local a, b, c = 3, 4, 5 -- un triplet de Pythagore
local i, arccos, exp = cpx.I, math.acos, cpx.exp
local f1 = function(z)
        return (z-c)*a/c*exp(-i*arccos(a/c))+c+i*c end
local M1 = matrixof(f1)
local f2 = function(z)
        return z*b/c*exp(i*arccos(b/c))+i*c end
local M2 = matrixof(f2)
local arbre
arbre = function(n)
    local color = mixcolor(ForestGreen,1,Brown,n)
    g:Linecolor(color); g:Dsquare(0,c,1,"fill=".color)
    if n > 0 then
        g:Savematrix(); g:Composematrix(M1); arbre(n-1)
        g:Restorematrix(); g:Savematrix(); g:Composematrix(M2)
        arbre(n-1); g:Restorematrix()
    end
end
arbre(8)
g:Show()
\end{luadraw}
```

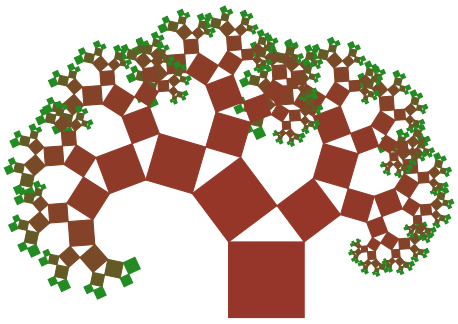

FIGURE 16 – Utilisation de la matrice du graphe

2.5 g:Rotate()

La méthode **g:Rotate(angle, center)** modifie la matrice de transformation du graphe *g* en la composant avec la matrice de la rotation d'angle *angle* (en degrés) et de centre *center*. L'argument *center* est un complexe qui vaut 0 par défaut.

2.6 g:Scale()

La méthode **g:Scale(factor, center)** modifie la matrice de transformation du graphe *g* en la composant avec la matrice de l'homothétie de rapport *factor* et de centre *center*. L'argument *center* est un complexe qui vaut 0 par défaut.

2.7 g:Savematrix() et g:Restorematrix()

- La méthode **g:Savematrix()** permet de sauvegarder dans une pile la matrice de transformation du graphe *g*.
- La méthode **g:Restorematrix()** permet de restaurer la matrice de transformation du graphe *g* à sa dernière valeur sauvegardée.

2.8 g:Setmatrix()

La méthode **g:Setmatrix(M)** permet d'affecter la matrice *M* à la matrice de transformation du graphe *g*.

2.9 g:Shift()

La méthode **g:Shift(v)** modifie la matrice de transformation du graphe *g* en la composant avec la matrice de la translation de vecteur *v* qui doit être un complexe.

```
\begin{luadraw}{name=free_art}
local du = math.sqrt(2)/2
local g = graph:new{window={1-du,4+du,1-du,4+du},
            margin={0,0,0,0},size={7,7}}
local i = cpx.I
g:Linestyle("noline")
g:Filloptions("full","Navy",0.1)
for X = 1, 4 do
    for Y = 1, 4 do
        g:Savematrix()
        g:Shift(X+i*Y); g:Rotate(45)
        for k = 1, 25 do
            g:Dsquare((1-i)/2,(1+i)/2,1)
            g:Rotate(7); g:Scale(0.9)
        end
        g:Restorematrix()
    end
end
g:Show()
\end{luadraw}
```

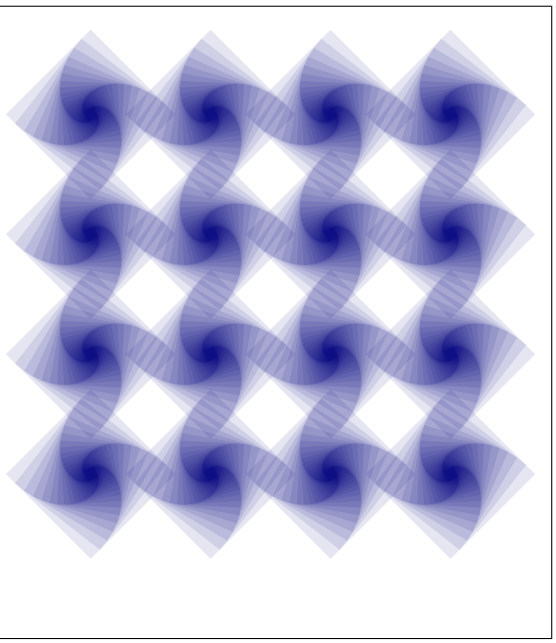

FIGURE 17 – Utilisation de Shift, Rotate et Scale

3) Changement de vue. Changement de repère

Changement de vue : lors de la création d'un nouveau graphique, par exemple :

```
local g = graph:new{window={-5,5,-5,5},size={10,10}}
```

L'option *window={xmin,xmax,ymin,ymax}* fixe la vue pour le graphique *g*, ce sera le pavé *[xmin, xmax]x [ymin, ymax]* de $\mathbf{R}^2$, et tous les tracés vont être clippés par cette fenêtre (sauf les labels qui peuvent débordés dans les marges, mais pas au-delà). Il est possible, à l'intérieur de ce pavé, de définir un autre pavé pour faire une nouvelle vue, avec la méthode **g:Viewport(x1,x2,y1,y2)**. Les valeurs de *x1, x2, y1, y2* se réfèrent la fenêtre initiale définie par l'option *window*. À partir de là, tout ce qui sort de cette nouvelle zone va être clippé, et la matrice du graphe est réinitialisée à l'identité, par conséquent il faut sauvegarder auparavant les paramètres graphiques courants :

```
g:Saveattr()
g:Viewport(x1,x2,y1,y2)
```

Pour revenir à la vue précédente avec la matrice précédente, il suffit d'effectuer une restauration des paramètres graphiques avec la méthode **g:Restoreattr()**.

Attention : à chaque instruction *Saveattr()* doit correspondre une instruction *Restoreattr()*, sinon il y aura une erreur à la compilation.

Changement de repère : on peut changer le système de coordonnées de la vue courante avec la méthode **g:Coordsystem(x1,x2,y1,y2,ortho)**. Cette méthode va modifier la matrice du graphe de sorte que tout se passe comme si la vue courante correspondait au pavé $[x1,x2]\times[y1,y2]$, l'argument booléen facultatif *ortho* indique si le nouveau repère doit être orthonormé ou non (false par défaut). Comme la matrice du graphe est modifiée il est préférable de sauvegarder les paramètres graphiques avant, et de les restaurer ensuite. Cela peut servir par exemple à faire plusieurs figures dans le graphique en cours.

```
\begin{luadraw}{name=viewport_changewin}
local g = graph:new{window={-5,5,-5,5},size={7,7}}
local i = cpx.I
g:Labelsize("tiny")
g:Writeln("\\tikzset{->-/.style={decoration={markings, mark=at
 ↪  position #1 with {\\arrow{>}}}, postaction={decorate}}}")
g:Dline({0,1},"dashed,gray"); g:Dline({0,i},"dashed,gray")
local legende = {"Point ordinaire", "Point d'inflexion",
 ↪  "Rebroussement 1ère espèce", "Rebroussement 2ème espèce"}
local A, B, C =(1+i)*0.75, 0.75, 0
local A2, B2 ={-1.25+i*0.5,-0.75-i*0.5,1.25-0.5*i, 0.5+i},
 ↪  {-0.75,-0.75,0.75,0.75}
local u = {Z(-5,0),Z(0,0),-5-5*i,-5*i}
for k = 1, 4 do
    g:Saveattr(); g:Viewport(u[k].re,u[k].re+5,u[k].im,u[k].im+5)
    g:Coordsystem(-1.4,2.25,-1,1.25)
    g:Composematrix({0,1,1+i}) -- pour pencher l'axe Oy
    g:Dpolyline({{-1,1},{-i*0.5,i}}) -- axes
    g:Lineoptions(nil,"blue",8)
    g:Dpath({A2[k],(B2[k]+2*A2[k])/3,(C+5*B2[k])/6,
     ↪  C,"b"},"->-=0.5")
    g:Dpath({C,(C+5*B)/6,(B+2*A)/3,A,"b"},"->-=0.75")
    g:Dpolyline({{0,0.75},{0,0.75*i}},false,"->,red")
    g:Dlabel(
        legende[k],0.75-0.5*i, {pos="S"},
        "$f^{(p)}(t_0)$",1,{pos="E",node_options="red"},
        "$f^{(q)}(t_0)$",0.75*i,{pos="W",dist=0.05})
    g:Restoreattr()
end
g:Show()
\end{luadraw}
```

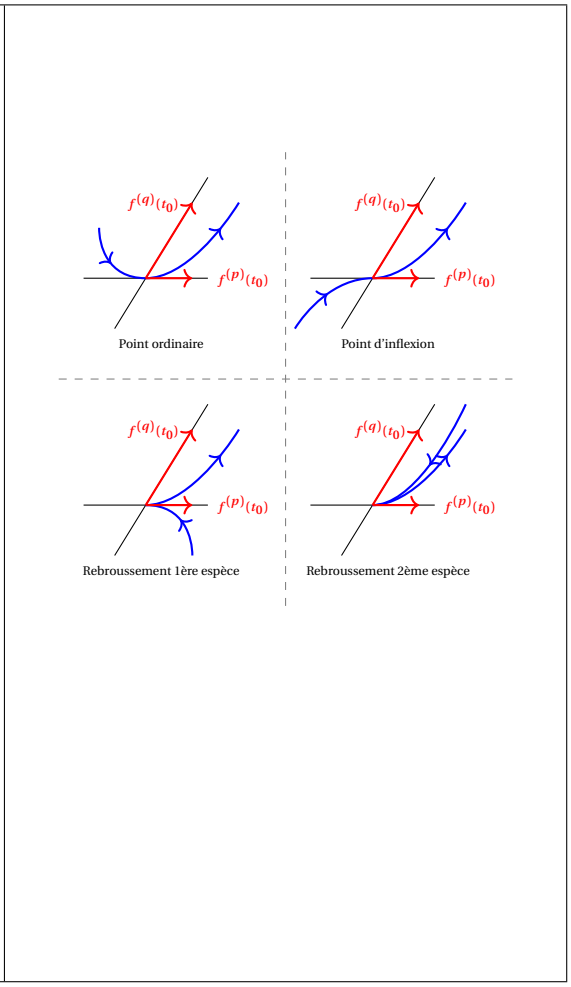

FIGURE 18 – Classification des points d'une courbe paramétrée

VI Ajouter ses propres méthodes à la classe *graph*

Sans avoir à modifier les fichiers sources Lua associés au paquet *luadraw*, on peut ajouter ses propres méthodes à la classe *graph*, ou modifier une méthode existante. Ceci n'a d'intérêt que si ces modifications doivent être utilisées dans différents graphiques et/ou différents documents (sinon il suffit d'écrire localement une fonction dans le graphique où on en a besoin).

1) Un exemple

Dans le graphique de la page 8, nous avons dessiné un champ de vecteurs, pour cela on a écrit une fonction qui calcule les vecteurs avant de faire le dessin, mais cette fonction est locale. On pourrait en faire une fonction globale (en enlevant le mot clé *local*), elle serait alors utilisable dans tout le document, mais pas dans un autre document !

Pour généraliser cette fonction, on va devoir créer un fichier Lua qui pourra ensuite être importé dans des documents en cas de besoin. Pour rendre l'exemple un peu consistant, on va créer un fichier qui va définir une fonction qui calcule les vecteurs d'un champ, et qui va ajouter à la classe *graph* deux nouvelles méthodes : une pour dessiner un champ de vecteurs

d'une fonction $f : (x,y) \to (x,y) \in \mathbf{R}^2$, on la nommera *graph:Dvectorfield*, et une autre pour dessiner un champ de gradient d'une fonction $f : (x,y) \to \mathbf{R}$, on la nommera *graph:Dgradientfield*. Du coup nous appellerons ce fichier : *luadraw_fields.lua*.

Contenu du fichier :

```
-- luadraw_fields.lua
-- ajout de méthodes à la classe graph du paquet luadraw
-- pour dessiner des champs de vecteurs ou de gradient
function field(f,x1,x2,y1,y2,grid,long)  -- fonction mathématique, indépendante du graphique
-- calcule un champ de vecteurs dans le pavé [x1,x2]x[y1,y2]
-- f fonction de deux variables à valeurs dans R^2
-- grid = {nbx, nby} : nombre de vecteurs suivant x et suivant y
-- long = longueur d'un vecteur
    if grid == nil then grid = {25,25} end
    local deltax, deltay = (x2-x1)/(grid[1]-1), (y2-y1)/(grid[2]-1) -- pas suivant x et y
    if long == nil then long = math.min(deltax,deltay) end -- longueur par défaut
    local vectors = {} -- contiendra la liste des vecteurs
    local x, y, v = x1
    for _ = 1, grid[1] do -- parcours suivant x
        y = y1
        for _ = 1, grid[2] do -- parcours suivant y
            v = f(x,y) -- on suppose que v est bien défini
            v = Z(v[1],v[2]) -- passage en complexe
            if not cpx.isNul(v) then
                v = v/cpx.abs(v)*long -- normalisation de v
                table.insert(vectors, {Z(x,y), Z(x,y)+v} ) -- on ajoute le vecteur
            end
            y = y+deltay
        end
        x = x+deltax
    end
    return vectors -- on renvoie le résultat (ligne polygonale)
end

function graph:Dvectorfield(f,args) -- ajout d'une méthode à la classe graph
-- dessine un champ de vecteurs
-- f fonction de deux variables à valeurs dans R^2
-- args table à 4 champs :
-- { view={x1,x2,y1,y2}, grid={nbx,nby}, long=, draw_options=""}
    args = args or {}
    local view = args.view or {self:Xinf(),self:Xsup(),self:Yinf(),self:Ysup()} -- repère utilisateur par défaut
    local vectors = field(f,view[1],view[2],view[3],view[4],args.grid,args.long) -- calcul du champ
    self:Dpolyline(vectors,false,args.draw_options) -- le dessin (ligne polygonale non fermée)
end

function graph:Dgradientfield(f,args) -- ajout d'une autre méthode à la classe graph
-- dessine un champ de gradient
-- f fonction de deux variables à valeurs dans R
-- args table à 4 champs :
-- { view={x1,x2,y1,y2}, grid={nbx,nby}, long=, draw_options=""}
    local h = 1e-6
    local grad_f = function(x,y) -- fonction gradient de f
        return { (f(x+h,y)-f(x-h,y))/(2*h), (f(x,y+h)-f(x,y-h))/(2*h) }
    end
    self:Dvectorfield(grad_f,args) -- on utilise la méthode précédente
end
```

2) Comment importer le fichier

Il y a deux méthodes pour cela :

1. Avec l'instruction Lua *dofile*. On peut l'écrire par exemple dans le préambule après la déclaration du paquet :

```
\usepackage[]{luadraw}
\directlua{dofile("<chemin>/luadraw_fields.lua")}
```

Bien entendu, il faudra remplacer `<chemin>` par le chemin d'accès à ce fichier.

L'instruction `\directlua{dofile("<chemin>/luadraw_fields.lua")}` peut être placée ailleurs dans le document pourvu que ce soit après le chargement du paquet (sinon la classe *graph* ne sera pas reconnue lors de la lecture du fichier). On peut aussi placer l'instruction `dofile("<chemin>/luadraw_fields.lua")` dans un environnement *luacode*, et donc en particulier dans un environnement *luadraw*.

Dès que le fichier est importé, les nouvelles méthodes sont disponibles pour la suite du document.

Cette façon de procéder a au moins deux inconvénients : il faut se souvenir à chaque utilisation de `<chemin>`, et d'autre part l'instruction *dofile* ne vérifie par si le fichier a déjà été lu. Pour ces raisons, on préférera la méthode suivante.

2. Avec l'instruction Lua *require*. On peut l'écrire par exemple dans le préambule après la déclaration du paquet :

```
\usepackage[]{luadraw}
\directlua{require "luadraw_fields"}
```

On remarquera l'absence du chemin (et l'extension lua est inutile).

L'instruction `\directlua{require "luadraw_fields"}` peut être placée ailleurs dans le document pourvu que ce soit après le chargement du paquet (sinon la classe *graph* ne sera pas reconnue lors de la lecture du fichier). On peut aussi placer l'instruction `require "luadraw_fields"` dans un environnement *luacode*, et donc en particulier dans un environnement *luadraw*.

L'instruction *require* vérifie si le fichier a déjà été chargé ou non, ce qui est préférable. Mais il faut cependant que Lua soit capable de trouver ce fichier, et le plus simple pour cela est qu'il soit quelque part dans une arborescence connue de TeX. On peut par exemple créer dans son *texmf* local le chemin suivant :

```
texmf/tex/lualatex/myluafiles/
```

puis copier le fichier *luadraw_fields.lua* dans le dossier *myluafiles*.

```
\begin{luadraw}{name=fields}
require "luadraw_fields" -- import des nouvelles méthodes
local g = graph:new{window={0,10,0,21},size={7,14}}
local i = cpx.I
g:Labelsize("footnotesize")
local f = function(x,y) return {x-x*y,-y+x*y} end -- Volterra
local F = function(x,y) return x^2+y^2+x*y-6 end
local H = function(t,Y) return f(Y[1],Y[2]) end
-- graphique du haut
g:Saveattr();g:Viewport(0,10,11,21);g:Coordsystem(-5,5,-5,5)
g:Dgradbox({-4.5-4.5*i,4.5+4.5*i,1,1},
{originloc=0,originnum={0,0},grid=true,title="gradient field,
 ↪  $f(x,y)=x^2+y^2+xy-6$"})
g:Arrows("->"); g:Lineoptions(nil,"blue",6)
g:Dgradientfield(F,{view={-4,4,-4,4},grid={15,15},long=0.5})
g:Arrows("-");g:Lineoptions(nil,"Crimson",12)
g:Dimplicit(F, {view={-4,4,-4,4}})
g:Restoreattr()
-- graphique du bas
g:Saveattr();g:Viewport(0,10,0,10);g:Coordsystem(-5,5,-5,5)
g:Dgradbox({-4.5-4.5*i,4.5+4.5*i,1,1},
  {originloc=0,originnum={0,0},grid=true,title="vector field,
   ↪  $f(x,y)=(x-xy,-y+xy)$"})
g:Arrows("->"); g:Lineoptions(nil,"blue",6)
g:Dvectorfield(f,{view={-4,4,-4,4}})
g:Arrows("-");g:Lineoptions(nil,"Crimson",12)
g:Dodesolve(H,0,{2,3},{t={0,50},out={2,3},nbdots=250})
g:Restoreattr()
g:Show()
\end{luadraw}
```

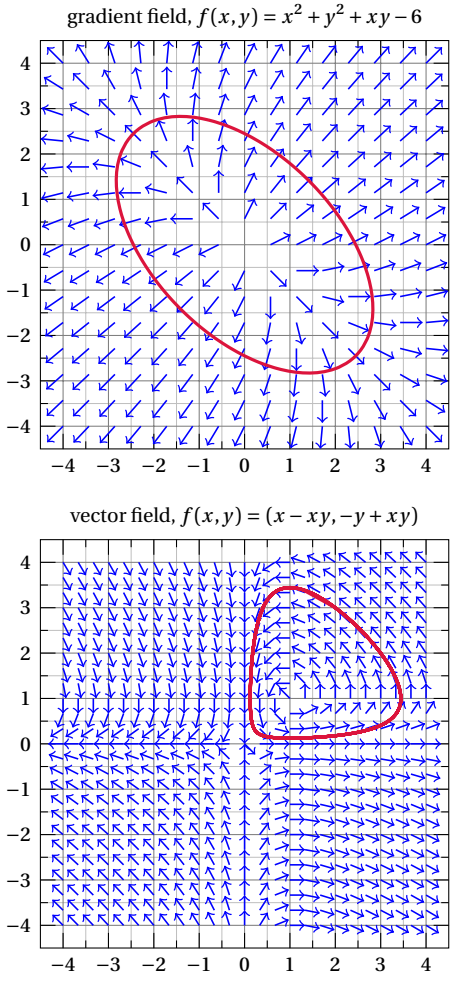

FIGURE 19 – Utilisation des nouvelles méthodes

3) Modifier une méthode existante

Prenons par exemple la méthode *DplotXY(X,Y,draw_options)* qui prend comme arguments deux listes (tables) de réels et dessine la ligne polygonale formée par les points de coordonnées (X[k],Y[k]). Nous allons la modifier afin qu'elle prenne en compte le cas où X est une liste de noms (chaînes), dans ce cas, on affichera les noms sous l'axe des abscisses (avec l'abscisse k pour le k^{e}nom) et on dessinera la ligne polygonale formée par les points de coordonnées (k,Y[k]), sinon on fera comme l'ancienne méthode. Il suffit pour cela de réécrire la méthode (dans un fichier Lua pour pouvoir ensuite l'importer) :

```
function graph:DplotXY(X,Y,draw_options)
-- X est une liste de réels ou de chaînes
-- Y est une liste de réels de même longueur que X
    local L = {} -- liste des points à dessiner
    if type(X[1]) == "number" then -- liste de réels
        for k,x in ipairs(X) do
            table.insert(L,Z(x,Y[k]))
        end
    else
        local noms = {} -- liste des labels à placer
        for k = 1, #X do
            table.insert(L,Z(k,Y[k]))
            insert(noms,{X[k],k,{pos="E",node_options="rotate=-90"}})
        end
        self:Dlabel(table.unpack(noms)) --dessin des labels
    end
    self:Dpolyline(L,draw_options) -- dessin de la courbe
end
```

Dès que le fichier sera importé, cette nouvelle définition va écraser l'ancienne (pour toute la suite du document). Bien entendu on pourrait imaginer ajouter d'autres options sur le style de tracé par exemple (ligne, bâtons, points ...).

```
\begin{luadraw}{name=newDplotXY}
require "luadraw_modified" -- import de la méthode modifiée
local g = graph:new{window={-0.5,11,-1,20}, margin={0.5,0.5,0.5,1},
 ↪  size={7,7,0}}
g:Labelsize("scriptsize")
local X, Y = {}, {} -- on définit deux listes X et Y, on pourrait
 ↪  aussi les lire dans un fichier
for k = 1, 10 do
    table.insert(X,"nom"..k)
    table.insert(Y,math.random(1,20))
end
defaultlabelshift = 0
g:Daxes({0,1,2},{limits={{0,10},{0,20}},
 ↪  labelpos={"none","left"},arrows="->", grid=true})
g:DplotXY(X,Y,"line width=0.8pt, blue")
g:Show()
\end{luadraw}
```

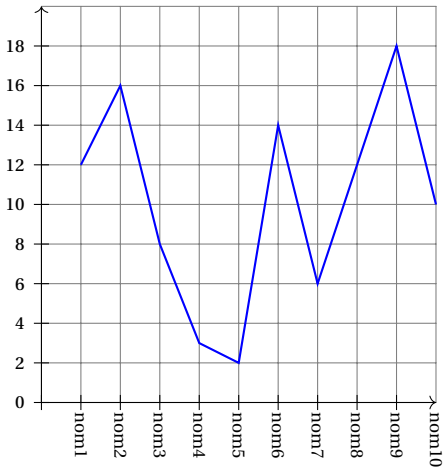

FIGURE 20 – Modification d'une méthode existante

VII Historique

1) Version 1.0

Première version.