Interaction entre algèbre linéaire et analyse en formalisation des mathématiques

Guillaume Cano

To cite this version:

Guillaume Cano. Interaction entre algèbre linéaire et analyse en formalisation des mathématiques. Autre [cs.OH]. Université Nice Sophia Antipolis, 2014. Français. NNT : 2014NICE4016 . tel-00986283

HAL Id: tel-00986283
https://theses.hal.science/tel-00986283

Submitted on 2 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE

pour l'obtention du grade de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Mention : Informatique

Présentée et soutenue par

Guillaume CANO

Interaction entre algèbre linéaire et analyse en formalisation des mathématiques

Thèse dirigée par Yves BERTOT

soutenue le 4 avril 2014

Jury :

Micaela MAYERO	-	Maître de conférence, LIPN (Rapporteur)
Julio RUBIO	-	Professeur, Université de la Rioja (Rapporteur)
Yves BERTOT	-	Directeur de recherche, INRIA (Directeur)
Stephen WATT	-	Professeur, Université Ontario Ouest (Examinateur)
Christine PAULIN-MOHRING	-	Professeur, Université Paris Sud (Examinateur)
Douglas HOWE	-	Professeur, Université Carlton (Examinateur)

Abstract

In this thesis we present the formalization of three principal results that are the Jordan normal form of a matrices, the Bolzano-Weierstraß theorem, and the Perron-Frobenius theorem.

To formalize the Jordan normal form, we introduce many concepts of linear algebra like block diagonal matrices, companion matrices, invariant factors, ...

The formalization of Bolzano-Weierstraß theorem needs to develop some theory about topological space and metric space.

The Perron-Frobenius theorem is not completly formalized. The proof of this theorem uses both algebraic and topological results. We will show how we reuse the previous results.

Résumé

Dans cette thèse nous présentons la formalisation de trois résultats principaux que sont la forme normale de Jordan d'une matrice, le théorème de Bolzano-Weierstraß et le théorème de Perron-Frobenius.

Pour la formalisation de la forme normale de Jordan nous introduisons différents concepts d'algèbre linéaire tel que les matrices diagonales par blocs, les matrices compagnes, les facteurs invariants, ...

Ensuite nous définissons et développons une théorie sur les espaces topologiques et métriques pour la formalisation du théorème de Bolzano-Weierstraß.

La formalisation du théorème de Perron-Frobenius n'est pas terminé. La preuve de ce théorème utilise des résultat d'algèbre linéaire, mais aussi de topologie. Nous montrerons comment les précédents résultats seront réutilisés.

Remerciements

Je voudrais dans un premier temps remercier Micaela Mayero et Julio Rubio pour avoir acceptés d'endurer l'épreuve d'être les rapporteurs de cette thèse.

Je remercie mon directeur de thèse Yves Bertot pour m'avoir permis de faire cette thèse, mais aussi pour les moments patisserie.

Je tiens à remercier aussi toute l'équipe Marelle pour son soutient. Je remercie Nathalie Bellesso qui est une secrétaire merveilleuse, qui m'a aidée dans mes tâches administratives et qui en plus écrit de bons livres. Je remercie José Grimm pour ces conseils TeXniques, Laurent Théry pour son soutient, ses conseils et ses casse-têtes. Et surtout un grand merci à Laurence Rideau qui a passé beaucoup de temps à m'apprendre SSReflect, à relire mes écrits, à corriger mes fautes réccurentes, qui a aussi eu la patience de me supporter, et aussi qui m'a encouragé, grondé et motivé tout au long de ma thèse. Je remercie également ceux qui ont passés du temps à la relecture de ma thèse, et que j'ai déjà cités.

Merci également à Ioana Paşca qi m'a beaucoup appris, en particulier une chose absolument fondamentale : le chocolat, au bureau, c'est important ! Merci à Tom Hutchinson dont j'ai apprécié la compagnie et les conversations. Merci aussi à Cyril Cohen, Maxime Dénès, Érik Martin-Dorel, Anders Mörtberg. Merci à ceux que j'ai oubliés.

Je remercie aussi ma famille pour son soutient à toute épreuve.

Et je voudrais rendre hommage à Pierre Damphousse, sans qui je n'aurais jamais fait cette thèse.

Table des matières

Introduction **1**
1 Preuve formelle 1
2 Assistants de preuve 2
3 Brève présentation de Coq 3
3.1 Les Types 4
4 Motivations 5
4.1 Motivations générales 5
4.2 Motivations de la thèse 6
5 Contributions 6

1 Contexte **9**
1.1 Les types de base 9
1.1.1 Le type produit 10
1.1.2 Les types propositionnels 10
1.1.3 Le type booléen 10
1.1.4 Les entiers naturels 11
1.1.5 Les séquences 11
1.2 La bibliothèque SSReflect 12
1.2.1 La réflexion 12
1.2.2 Le système de vue 14
1.2.3 Les types avec une égalité décidable 15
1.2.4 Les types finis 16
1.2.5 Les opérateurs de familles 16
1.2.6 Les structures algébriques 18
1.2.7 Les polynômes 20
1.2.8 Les matrices 21
1.2.9 Matrices et polynômes 22
1.2.10 Les structures ordonnées 23
1.2.11 Les nombres complexes 24
1.3 Conclusion 24

2 Forme de Jordan d'une matrice **25**
2.1 Matrices diagonales par blocs 25
2.1.1 Définition idéale 26

2.1.2 Problèmes en dimension nulle 29
2.1.3 Définition finale 29
2.2 Matrices semblables et équivalentes 31
2.2.1 Définitions 31
2.2.2 Théorème fondamental 32
2.3 Forme normale de Smith 33
2.3.1 Spécification 34
2.3.2 Unicité 35
2.3.3 Facteurs invariants 39
2.4 Forme normale de Frobenius 40
2.4.1 Matrices compagnes 40
2.4.2 De Smith à Frobenius 42
2.5 Forme normale de Jordan 44
2.5.1 Définitions 45
2.5.2 De Frobenius à Jordan 45
2.5.3 Diagonalisation 48
2.6 Conclusion 49
2.7 Travaux reliés 50

3 Topologie **51**
3.1 Théorie des ensembles 51
3.1.1 Réécriture 52
3.1.2 Les familles d'ensembles 53
3.2 Structures Topologiques 55
3.2.1 Définition 55
3.2.2 Approche naïve 57
3.2.3 Approche SSReflect 59
3.3 Bolzano-Weierstraß 61
3.3.1 Topologie générale 62
3.3.2 Les suites 62
3.3.3 Le théorème 63
3.4 Conclusion 66
3.5 Travaux reliés 67

4 Perron-Frobenius **69**
4.1 Rappels et définitions 70
4.1.1 Minimum et maximum 70
4.1.2 Éléments d'algèbre linéaire 71
4.2 Matrices strictement positives 73
4.2.1 Preuve du théorème principal 73
4.2.2 Instance des structures topologiques 75
4.2.3 Preuve de l'axiome 77
4.3 Matrices à coefficients positifs ou nuls 80
4.3.1 La preuve 81

4.3.2 Convergence des rayons spectraux 81
4.3.3 Compacité de la boule unité 83
4.4 Conclusion 83
4.5 Travaux reliés 84

Conclusion **85**

Bibliographie **87**

Introduction

1 Preuve formelle

Formaliser un théorème, c'est difficile à définir formellement. Dans le mot "formelle" on a le mot "forme". Une preuve formelle, serait donc une preuve qui manipule seulement des formes, des symboles, en faisant abstraction du sens que peuvent avoir ces symboles.

Le mieux serait d'essayer de comprendre sur un exemple. Essayons de formaliser le résultat suivant : *"Si nous avons un ensemble, avec une loi associative sur cet ensemble qui possède un élément neutre, et si un élément de l'ensemble possède un inverse à droite et un inverse à gauche, alors l'inverse à droite et l'inverse à gauche sont égaux".*

Nous commencons d'abord par *mettre en forme* l'énoncé, c'est-à-dire l'exprimer à l'aide de symboles. Nous pouvons utiliser la lettre "E" pour désigner l'ensemble, le symbole "$*$" pour la loi, et "e" pour son élément neutre. Nous pouvons formaliser le fait que la loi est associative et que e et un élément neutre par les hypothèses suivantes :

hA : $\forall xyz \in E, (x * y) * z = x * (y * z)$.

n1 : $\forall x \in E, x * e = x$.

n2 : $\forall x \in E, e * x = x$.

Ensuite nous pouvons utiliser par exemple la lettre "a" pour désigner un élément de E qui a un inverse à gauche et à droite que nous appellerons $a1$ et $a2$. Formellement, nous avons les deux hypothèses suivantes :

ha1 : $a * a1 = e$

ha2 : $a2 * a = e$

et donc nous voulons prouver que $a1 = a2$.

Nous avons pour l'instant décrit formellement les hypothèses de l'énoncé, ainsi que le but que nous voulons prouver. Nous pouvons maintenant prouver formellement que $a1 = a2$ en utilisant uniquement les hypothèses que nous avons formalisées.

Nous avons donc $a1 = e * a1 = (a2 * a) * a1 = a2 * (a * a1) = a2 * e = a2$, en utilisant respectivement les hypothèses n2, ha2, hA, ha1, n1. Pour vérifier cette preuve, nous n'avons pas besoin de connaitre le sens des symboles. La première hypothèse utilisée est l'hypothèse n2. Si on regarde cette hypothèse, on s'aperçoit qu'il y a un signe "=" qui indique que la *forme* qu'il y a d'un coté de ce signe est équivalente à la *forme* qu'il y a de l'autre coté du signe, ainsi dans une expression nous pouvons remplacer l'une par l'autre. Donc l'hypothèse n2 dit que le dessin "$a1$" a la même *forme* que le dessin "$e * a1$". On n'a pas besoin de savoir à quoi correspond $*$ ou e, on a seulement besoin de savoir que l'on peut remplacer une expression par une autre, et donc on peut vérifier la preuve juste en comparant des dessins[1]. C'est ce qu'on appelle une preuve formelle.

Cette abstraction du sens des symboles que l'on manipule fait qu'une preuve formelle est généralement plus longue et détaillée qu'une preuve *normale*. En effet, dans une preuve mathématique, certaines choses sont considérées comme évidentes. Mais si l'on exprime ces choses à l'aide de symboles, et que l'on oublie le sens de ces symboles, alors en gros la seule trivialité qui reste est $x = x$.

D'un autre côté, le fait de pouvoir faire des preuves en manipulant des symboles sans se préoccuper du sens fait que certaines choses peuvent se faire de manière automatique, et donc être faites par une machine.

2 Assistants de preuve

Un assistant de preuve est un logiciel permettant de faire des preuves formelles. Il en existe de deux sortes :

- Les assistants de preuve automatiques : ce sont des logiciels qui prouvent automatiquement les théorèmes. L'exemple utilisé ci-dessus serait très rapidement prouvé par un prouveur automatique. Pour prouver les théorèmes, le logiciel regarde parmi la liste des théorèmes qu'il a à sa disposition, lesquels il peut appliquer. Il essaye ensuite les lemmes qu'il peut appliquer. Pour chaque théorème appliqué, les conditions du théorème deviennent de nouveaux buts à prouver, il génère ainsi une sorte d'arbre de preuve. Si une branche de cet arbre termine, alors il annonce que le théorème est valide, sinon nous n'avons aucune information sur la validité du résultat que nous voulions prouver. Si l'on donne un théorème à un prouveur automatique, il répondra soit "c'est vrai" soit "je ne sais pas", mais il n'indiquera pas comment il a prouvé le théorème. On peut cependant demander une trace de ce qu'il a fait, mais c'est rarement exploitable, on se sert géneralement de la trace pour voir quels théorèmes intermédiaires manquent à l'assistant de preuve pour valider la preuve.

[1]Enfin pas tout à fait. Dans cet exemple, il faut au moins connaître les symboles $\forall$, $\in$, et $=$. Pour avoir une preuve formelle complète, il aurait fallu donner aussi les règles d'utilisation de ces symboles.

Ceci est une présentation assez grossière des assistants de preuve automatiques, on pourra trouver dans [6] ou dans [24] plus de détails sur les mécanismes utilisés. Les principaux prouveurs automatiques[1] sont ACL2 [28, 29], Alt-Ergo [5], E-prover [44], Gappa [31], Z3 [11], ...

- Les assistants de preuve semi-automatiques : contrairement aux assistants de preuve automatiques, les preuves sont faites par un utilisateur humain. L'utilisateur dit quel théorème il veut utiliser, et la machine indique si oui ou non le théorème peut être appliqué ou pas. Donc ici l'humain fait la preuve, et la machine la vérifie. Parmi les principaux assistants de preuve semi-automatique, on trouve HOL [35], HOL-Light [25], Isabelle [50], Coq [13], Mizar [33], Matita [1], PVS [45], Nuprl [10] ...

  Dans cette thèse nous utiliserons l'assistant de preuve Coq.

3 Brève présentation de Coq

Le logiciel Coq est un assistant de preuve semi-automatique qui permet de faire des preuves, mais aussi des programmes (et des preuves sur ces programmes !). La construction de preuves se fait à l'aide d'un langage de tactiques. Une tactique c'est ce qui permet de dire au système quel règle logique utiliser pour construire les preuves.

La formalisation des théorèmes en Coq se fait de manière interactive avec la machine. En général nous avons un environnement de preuve qui se présente comme suit :

```
       H
===================
       B
```

où `H` est la partie où sont nos hypothèses, et `B` est la partie où il y a le(s) but(s) que nous voulons prouver. Supposons par exemple que nous ayons l'environnement de preuve suivant :

```
    ⋮
Hn0 : n = 0
H : m = n -> P
===================
 P
```

Nous pouvons écrire dans le script de preuve que l'on veut appliquer l'hypothèse `H` en écrivant :

```
apply: H.
```

Nous pouvons ensuite exécuter la commande que l'on vient d'écrire pour voir comment notre environnement de preuve évolue. Ici il y a deux possibilités, soit

[1] On peut trouver une liste plus complète dans http://why3.lri.fr/.

le système Coq ne peut pas appliquer l'hypothèse et dans ce cas il renvoie un message d'erreur, soit comme c'est le cas ici, le système Coq peut appliquer l'hypothèse et dans ce cas il va alors demander de prouver les conditions de l'hypothèse. Nous aurons donc le nouvel environnement de preuve suivant :

```
    ⋮
Hn0 : n = 0
===================
 m = n
```

Là, nous savons que $n = 0$, nous pouvons donc remplacer n par 0 dans notre but. Pour cela, nous pouvons écrire dans le script de preuve la ligne suivante :

```
rewrite Hn0.
```

Après l'exécution de cette ligne, nous aurons l'environnement de preuve suivant :

```
    ⋮
Hn0 : n = 0
===================
 m = 0
```

Dans cet exemple, `apply` et `rewrite` sont ce qu'on appelle des tactiques. Au départ nous voulions prouver la proposition `P`, et en utilisant les deux tactiques ci-dessus, nous nous somme ramenés à prouver `m = 0`. C'est en utilisant les tactiques et nos hypothèses que nous allons construire progressivement la preuve formelle.

Le langage que nous utiliserons pour la formalisation des théorèmes s'appelle SSReflect [20]. C'est une extension de Coq créée par Georges Gonthier pour prouver le théorème des quatre couleurs [18]. Le langage de tactique est défini différemment mais le noyau qui vérifie les preuves reste le même. Cette bibliothèque possède déjà les outils de base pour faire de l'algèbre linéaire.

3.1 Les Types

Coq est un langage fortement typé. Un type définit un ensemble d'objets de même nature. Par exemple dans Coq tous les entiers naturels sont de type `nat`, on écrit :

```
3 : nat
```

et ça se lit "trois est de type nat".

Le type `Prop` est le type des propositions, par exemple :

```
(forall n : nat, n + 0 = n) : Prop
```

Maintenant dire que le type de `H` est la proposition `forall n : nat, n + 0 = n`, c'est dire que `H` est une preuve de cette proposition. Si on reprend l'exemple ci-dessus, on a dans nos hypothèses la ligne suivante :

```
Hn0 : n = 0
```

Cela signifie que parmi nos hypothèses on a une preuve (ici appelé `Hn0`) que `n = 0`. Toutes les preuves d'une proposition ont le même type.

Le type d'une fonction est indiqué par une flèche. Le type `nat -> nat`, par exemple, est le type d'une fonction qui prend en argument un entier et qui retourne comme résultat un entier. On utilise la même flèche pour l'implication. En effet, si `P` et `Q` sont des propositions, `P -> Q` est le type d'une fonction qui prend en argument une preuve de `P` et qui retourne comme résultat une preuve de `Q`, c'est bien une preuve de `P` implique `Q`.

Cette correspondance entre les types et les preuves s'appelle la correspondance de Curry-Howard. Elle fait le lien entre une preuve d'une proposition `P`, et le type d'un terme de λ-calcul. Donc vérifier une preuve revient à vérifier le type d'un λ-terme. Et la vérification de type est quelque chose qui peut se faire de manière automatique par un ordinateur. Coq est un assistant de preuve qui utilise cette correspondance.

Coq permet également de travailler avec des types dépendants. Un type dépendant est un type qui dépend de un ou plusieurs paramètres. Par exemple, le type d'une matrice est un type qui dépend du nombre de lignes et de colonnes de la matrice.

4 Motivations

4.1 Motivations générales

Il y a deux principales raisons pour lesquelles on voudrait faire des preuves formelles. La première est pour vérifier des programmes. Par exemple, les programmes utilisés dans les métros automatiques qui n'ont pas de conducteur, ou les programmes utilisés dans des avions, etc. C'est le genre de programmes dont on voudrait être sûr qu'ils ne feront pas d'erreurs.

La seconde, est que certaines preuves mathématiques nécessitent de longs calculs par ordinateur. C'est le cas par exemple du théorème des quatre couleurs ou dernièrement de la conjecture de Kepler qui est en train d'être formalisée avec l'assistant de preuve HOL Light [21, 22]. Donc prouver formellement que les calculs faits par l'ordinateur sont corrects permet d'avoir une meilleure certitude sur ces preuves.

Il y a aussi le fait que les preuves mathématiques deviennent plus difficiles. Il existe des théorèmes dont la preuve n'est compréhensible que seulement par une poignée de mathématiciens. Ce sont des théorème difficiles à vérifier et prouver formellement ces théorèmes permettrait de pouvoir vérifier ces preuves de manières automatiques. C'est le cas par exemple du théorème de Feit-Thompson [19] dont la preuve fait plusieurs centaines de pages et qui a été formalisé avec l'assistant de preuve Coq.

4.2 Motivations de la thèse

Les démonstrations de certains théorèmes mathématiques parfois font intervenir des résultats de différentes branches des mathématiques. C'est le cas par exemple du théorème de Perron-Frobenius qui parle des propriétés du rayon spectral des matrices à coefficients positifs.

La preuve de ce théorème utilise des résultats d'algèbre linéaire, comme la réduction de Jordan d'une matrice, mais aussi des résultats d'analyse comme le théorème de Bolzano-Weierstraß en topologie générale, ou des résultats de convergence sur les matrices.

Le but de cette thèse a été d'essayer de fournir des outils d'algèbre linéaire et d'analyse assez généraux pour qu'ils puissent être réutilisés.

Nous présenterons dans cette thèse d'abord les travaux sur la forme normale de Jordan, ensuite ceux sur la topologie générale. Ces deux parties ont été formalisées indépendamment l'une de l'autre. Nous verrons une réutilisation de ces résultats dans le dernier chapitre avec le théorème de Perron-Frobenius.

Ce qui a amené en premier lieu à s'intéresser au théorème de Perron-Frobenius est un problème de robotique qui a demandé à étudier des matrices dont les coefficients sont des intervalles [32]. Le fait que l'on ait besoin du théorème de Perron-Frobenius a été mis en évidence par Ioana Paşca en étudiant la régularité de ces matrices [42].

5 Contributions

La preuve du théorème de Perron-Frobenius qui a été formalisée en partie dans cette thèse utilise deux résultats fondamentaux en mathématiques qui sont le théorème de Bolzano-Weierstraß en topologie générale et la forme normale de Jordan d'une matrice en algèbre linéaire.

Jusqu'à présent les notions de topologie telles que la continuité de fonction, ou la convergence de suite, par exemple, sont formalisées dans un cadre restreint, c'est-à-dire la plupart du temps celui des nombres réels. Pour la formalisation du théorème de Perron-Frobenius, nous avons besoin d'utiliser des propriétés de topologie sur des matrices. C'est pourquoi cette thèse propose une formalisation de résultats de topologie générale qui pourront être utilisés pour n'importe quel type, du moment que l'on aura instancié une topologie sur ce type. Un des résultats important de cette formalisation est le théorème de Bolzano-Weierstraß.

La formalisation de la forme normale de Jordan a, entre autre, apporté une formalisation des matrices diagonales par blocs, d'une théorie sur les matrice semblables et équivalentes, de la théorie des facteurs invariants, des matrices compagnes, en formalisant au passage la forme normale de Frobenius. En collaboration avec Maxime Dénès, nous avons pu, en utilisant les outils fournis par les bibliothèques CoqEAL[1] et SSReflect, faire une formalisation complète (sans

[1] http://www.maximedenes.fr/content/coqeal-coq-effective-algebra-library

axiomes) de ces résultats.

Pour la preuve du théorème de Perron-Frobenius, nous avons formalisé des théorèmes généraux sur la convergence des suites sur des structures ordonnées et archimédiennes, mais aussi sur la convergence des suites de matrices. En particulier nous établissons formellement quelles sont les conditions nécessaires et suffisantes pour que la suite des puissances d'une matrice converge vers zéro. Nous donnons également une définition formelle du rayon spectral d'une matrice.

Pour résumer, les contributions originales de cette thèse sont :

- Une théorie générale sur la topologie.
  - dont le théorème de Bolzano-Weierstraß.
- Les matrices diagonales par blocs.
- Une théorie des facteurs invariants.
- Les matrices compagnes.
- Les formes normales de Frobenius et de Jordan d'une matrice.
- Des théorèmes généraux sur la convergence des suites
  - sur les structures ordonnées.
  - sur les structures archimédiennes.
  - sur les matrices.
- Une définition du rayon spectral d'une matrice.
- Une preuve partielle du théorème de Perron-Frobenius.

Chapitre 1

Contexte

L'outil utilisé pour la formalisation de tous les résultats présentés dans cette thèse est l'assistant de preuve Coq avec la bibliothèque SSReflect. La bibliothèque SSReflect contient déjà des théories assez développées sur les principaux concepts que nous utiliserons tels que les polynômes ou les matrices par exemple. Nous présenterons donc ici comment les principaux outils que nous utiliserons par la suite sont formalisés dans la bibliothèque SSReflect, afin de familiariser le lecteur avec les notations et les énoncés formels qu'il rencontrera.

1.1 Les types de base

Les notions abordées dans cette section sont expliquées plus en détail dans [3].

La plupart des types en Coq sont des types inductifs. Les types inductifs sont en général définis comme suit :

```
Inductive TI := C1 : e1 | C2 : e2 | ... | Ck : ek.
```

Ici les `ei` sont des expressions dans lesquelles le type `TI` peut apparaître. Les `Ci` sont les constructeurs du type `TI`. C'est ce qui permet de construire des éléments de type `TI`. Tous les éléments de type `TI` sont forcément construits à partir des constructeurs.

Nous pouvons définir des fonctions du type inductif `TI` vers un type `T` en faisant un traitement par cas sur toutes les formes possibles de l'élément de type `TI` passé en argument de la fonction :

```
Definition f (x : TI) : T :=
  match x with
    |C1 ... => E1
    |C2 ... => E2
      ⋮
    |Ck ... => Ek
  end.
```

où les points de suspensions horizontaux sont de possibles arguments des constructeurs du type `TI`. L'expression `f (Ci ...)` est convertible avec l'expression `Ei`, c'est-à-dire que l'expression `f (Ci ...)` se réduit en `Ei` par des règles de calculs. On peut considérer deux expressions convertibles comme étant identiques.

Nous allons voir dans la suite les types inductifs de base dans Coq.

1.1.1 Le type produit

Le type produit de deux types `A` et `B` est défini comme suit :

```
Inductive prod (A B : Type) :=
  pair : A -> B -> prod A B.
```

Ce type correspond au produit cartésien de `A` et de `B`. La notation définie dans la bibliothèque standard de Coq pour `prod A B` est `A * B`, et celle pour `pair x y` est `(x,y)`. Donc si nous avons `x : A` et `y : B` alors nous avons `(x,y) : A * B`.

Si nous avons un élément `p : A * B` alors on peut définir les projections qui permettent de récupérer le premier et le second élément de la paire `p` :

```
Definition fst (A B : Type) (p : A * B) :=
  match p with
   (x,y) => x
  end.
```

La fonction `snd` qui renvoie le second élément de `p` est définie de la même manière. Dans la bibliothèque SSReflect, la notation `p.1` désigne `fst p` et `p.2` désigne `snd p`. Donc `(x,y).1 = x` et `(x,y).2 = y`.

1.1.2 Les types propositionnels

Soit A et B deux propositions, alors les propositions « *A et B* » et « *A ou B* » sont exprimées formellement à l'aide de types inductifs. Elles sont notées respectivement `A /\ B` et `A \/ B`.

Pour la conjonction de plusieurs propositions par exemple, on peut rencontrer dans la bibliothèque SSReflect, la notation suivante :

```
[/\ A , B & C]
```

Pour la disjonction le `/\` est remplacé par `\/` et le `&` par `|`. On peut utiliser cette notation jusqu'à cinq propositions pour la conjonction, et quatre pour la disjonction.

1.1.3 Le type booléen

Les booléens sont définis simplement dans Coq de la manière suivante :

```
Inductive bool := true : bool | false : bool.
```

Les opérateurs booléens « et » et « ou » sont notés respectivement `a && b` et `a || b` pour deux booléens `a` et `b` quelconques.

1.1.4 Les entiers naturels

Les entiers naturels sont définis selon la construction de Peano :

```
Inductive nat :=
    O : nat
  | S : nat -> nat.
```

Le constructeur `S` correspond à la fonction *successeur*. Dans la bibliothèque SSReflect, `S n` est noté `n.+1`, `n.+2` est une notation pour `S n.+1`, cela va jusqu'à `n.+4`.

La fonction prédécesseur d'un entier est définie dans la bibliohèque standard de Coq :

```
Definition pred n := match n with 0 => 0 | S n' => n' end.
```

La bibliothèque SSReflect fourni pour `pred n` une notation similaire à la précedente qui est `n.-1`, de même nous avons aussi la notation `n.-2`.

1.1.5 Les séquences

Une séquence permet d'énumérer un nombre fini d'éléments de même type. On utilise plus couramment le mot « liste », d'ailleurs dans la bibliothèque standard de Coq ce type s'appelle `list`. Pour certaines raisons historiques dans la bibliothèque SSReflect on utilise la notation `seq` pour `list`, donc bien que la définition des séquences soit définie dans la bibliothèque standard de Coq, je vais la présenter ici avec la notation SSReflect car c'est celle que nous utiliserons par la suite :

```
Inductive seq (T : Type) :=
   nil : seq T
 | cons : T -> seq T -> seq T.
```

Les notations et les quelques définitions que nous allons voir maintenant sont celles de la bibliothèque SSReflect.

Le constructeur `nil`, noté `[::]`, correspond à la séquence vide. Le constructeur `cons` permet d'ajouter un élément en tête d'une séquence. Par exemple si `s` est la séquence qui contient (dans cet ordre) les éléments 2 et 3 alors `cons 1 s` est la séquence qui contient (dans cet ordre) les éléments 1, 2 et 3. On note `cons a s` par `a :: s`, par exemple la séquence ci-dessus est notée `1 :: (2 :: (3 :: nil))`. Une autre façon de noter cette séquence est la suivante `[:: 1 ; 2 ; 3 ]`.

Voici plusieurs fonctions élémentaires sur les séquences :

- La fonction `size` retourne le nombre d'éléments d'une séquence.

- La fonction `nth` permet de retourner le n-ième élément d'une séquence. Cette fonction prend un élément par défaut comme argument. Par exemple `nth x0 s i` retourne `x0` si la séquence `s` est vide ou si l'entier `i` est plus grand que `size s` et retourne le i-ème élément de la séquence sinon. Si le

type des éléments d'une séquence `s` a une structure de groupe additif avec comme élément neutre `0`, alors la notation `s`_i` désigne `nth 0 s i`.

- la fonction `last` retourne le dernier élément d'une séquence. Par exemple, `last x0 s` retourne le dernier élément de la séquence `s` et `x0` si la séquence `s` est vide.

- La fonction `cat` permet de concaténer deux séquences. `cat s1 s2` se note `s1 ++ s2`.

- La fonction `map` permet à partir d'une séquence `s` et d'une fonction `f` d'obtenir une nouvelle séquence en appliquant `f` à chaque élément de `s`. Par exemple `map f [:: 1 ; 2 ; 3]` est la séquence `[:: f 1 ; f 2 ; f 3]`. La notation `[seq f x | x <- s]` est une notation dite par compréhension pour `map f s`. On peut lire cette notation comme « La séquence des `f x` tels que `x` appartient à `s` »(le signe `<-` symbolise $\in$).

- La fonction `filter` permet à partir d'un prédicat booléen `P` et d'une séquence `s` d'obtenir une séquence qui ne contient que les éléments de la séquence `s` qui vérifient le prédicat `P`. La notation par compréhension de `filter P s` est `[seq x <- s | P x]`.

1.2 La bibliothèque SSReflect

Dans cette section, nous allons expliquer dans un premier temps quelques mécanismes utilisés dans la bibliothèque SSReflect. Nous présenterons ensuite les principaux outils de SSReflect utilisés pour la formalisation.

1.2.1 La réflexion

La réflexion est une manière d'utiliser le calcul pour prouver un résultat. Imaginez un miroir. D'un côté du miroir il y a le monde des propositions, et de l'autre côté il y a le monde des booléens. Si certaines propositions (pas toutes) se regardent dans le miroir, elle pourront voir une expression booléenne qui reflète leur comportement. C'est-à-dire que l'expression booléenne aura la valeur `true` si la proposition est vraie et la valeur `false` sinon. Prouver une proposition `P` par réflexion, c'est passer de l'autre côté du miroir, pour calculer la valeur de l'expression booléenne et ainsi en déduire la valeur de vérité de la proposition.

Dans la bibliothèque SSReflect, les lemmes qui permettent de passer d'un côté ou de l'autre du miroir sont appelés des lemmes de réflexion. Ces lemmes s'expriment à l'aide du type inductif suivant :

```
Inductive reflect (P : Prop) : bool -> Set :=
   ReflectT (_ : P) : reflect P true
 | ReflectF (_ : ~P) : reflect P false.
```

Le type `reflect` a deux constructeurs de type légèrement différent. Le premier constructeur prend une preuve de la proposition `P` en argument (cette preuve n'étant pas utilisée dans le type du constructeur, il est inutile de lui donner un nom, d'où le `_:`), et son type est `reflect P true`. Alors que le deuxième constructeur prend en argument une preuve de `~P` et a comme type `reflect P false`. Nous avons donc une correspondance entre une preuve de `P` avec la valeur `true`, et une preuve de `~P` avec la valeur `false`.

Si `H` est de type `reflect P b` alors nous avons deux cas. Soit `H = ReflectT H'`, et dans ce cas nous avons d'une part `b = true` (car le type de `ReflectT H'` est `reflect P true`), et d'autre part la proposition `P` est vraie (car `H'` est une preuve de `P`). Soit `H = ReflectF H'` et alors dans ce cas `b = false` et `H'` est une preuve de `~P`. Ainsi `H : reflect P b` exprime bien le fait qu'il y a une réflexion entre la valeur de `b` et la proposition `P`.

Un lemme de réflexion sur la proposition `P` permet de faire un raisonnement classique sur cette proposition. Si nous avons l'environnement de preuve suivant :

```
  ⋮
 H : reflect P b.
==========================
  G(b)
```

alors nous pouvons faire un traitement par cas sur l'hypothèse `H` :

```
case: H.
```

Cette tactique va essayer de déstructurer l'hypothèse `H`. Comme le type de `H` est inductif et possède deux constructeurs, cela va générer deux sous-buts (un pour chaque constructeur). Chacun des constructeurs ayant besoin d'un argument, chaque sous-but va être quantifié sur ces arguments ; dans le premier cas l'argument est une preuve de `P`, et dans le deuxième cas c'est une preuve de `~P`. Nous allons donc avoir les deux sous-buts suivants :

```
subgoal 1 :
 ⋮
==============================
   P -> G(true)

subgoal 2 is :
  ~P -> G(false)
```

Mis à part le remplacement du booléen `b` par sa valeur dans chacun des cas, nous obtenons le même résultat que si l'on avait utilisé la tactique `case: HP`, où `HP` est une hypothèse de type `P \/ ~P`.

Avec des lemmes de réflexion il est possible de passer facilement du monde des propositions au monde booléen (et vice versa) grâce au mécanisme de « vue » que nous allons présenter dans la section suivante.

1.2.2 Le système de vue

Dans la bibliothèque SSReflect, il existe un mécanisme de « vue ». Il est nommé ainsi car il permet de faire, dans un certain sens, des changements de point de vue.

Pour utiliser le système de vue on utilise la syntaxe suivante :

```
tactic/H.
```

où `tactic` peut être une tactique quelconque[1], et `H` une hypothèse ayant une des trois formes suivantes :

1. `P -> Q`
2. `P <-> Q`
3. `reflect P b`

où `P, Q` sont des propositions et `b` un booléen.

Dans chaque cas, il y a deux exemples d'utilisation des vues. Regardons d'abord les cas 1) et 2) :

- Le premier exemple d'utilisation est quand le but que l'on veut prouver est `Q`. Alors dans ce cas là la tactique `apply/H` transforme le but en `P`.
- Le deuxième exemple d'utilisation est quand le but que l'on veut prouver est de la forme `P -> G` alors la tactique `move/H` transforme le but en `Q -> G`.

La différence entre le cas 1) et le cas 2) est que dans le cas 2) les rôles des propositions `P` et `Q` peuvent être inversés.

Nous allons expliquer maintenant le cas 3). Dans la bibliothèque SSReflect, il y a une coercition du type des booléens vers le type des propositions :

```
Coercion is_true (b : bool) := b = true.
```

Cette coercition permet d'utiliser les booléens comme des propositions. Donc dans le cas où `H` est de la forme `reflect P b`, le système de vues se comporte comme si l'on avait `H : P <-> b` (i.e `P <-> b = true`).

Expliquons le principe sur un exemple. Imaginons que l'on veuille prouver qu'un booléen `b` soit égal à `true`, et que l'on ait une hypothèse `H : reflect P b`. Nous avons donc l'environnement de preuve suivant :

```
  ⋮
H : reflect P b
========================
  b
```

Si l'on exécute la tactique `apply/H` nous obtiendrons :

[1] Enfin presque, disons plutôt à quelque exceptions près. Par exemple on ne peut pas utiliser de vue avec la tactique `rewrite`.

```
  ⋮
H : reflect P b
=======================
  P
```

Pour voir ce qui s'est passé, nous pouvons exécuter la commande `Show Proof`, le système affichera un terme de preuve dans lequel on verra apparaître :

```
... [eta introTF H] ...
```

Donc lorsque nous avons voulu appliquer une vue avec l'hypothèse `H`, le système a fait appel à un lemme appelé `introTF`. Le lemme `introTF` est défini dans la bibliothèque SSReflect comme suit :

```
Lemma introTF (P : Prop) (b c : bool) : reflect P b ->
  (if c then P else ~P) -> b = c.
```

Le but que l'on veut démontrer est `b = true`, donc si nous appliquons le lemme `introTF` il va être instancié avec `c = true`, et le système va nous demander de prouver `if true then P else ~P` (c'est-à-dire `P`) et `reflect P b`, mais cette dernière hypothèse est donnée par `H`. Donc finalement le nouveau but sera transformé en `P`.

Le théorème `introTF` a été ajouté dans une base de données, que nous appelerons la base des lemmes de vues, à l'aide de la commande `Hint View`. Donc lorsque nous avons exécuté la tactique `apply/H`, le système a cherché s'il y a un lemme dans la base des lemmes de vues qui pourrait être utilisé avec l'hypothèse `H`. Dans notre cas il a trouvé le lemme `introTF`, s'il n'avait pas trouvé de lemme qu'il aurait pu utiliser, il aurait essayé d'appliquer directement l'hypothèse `H` (c'est ce qu'il fait dans le cas où `H : P -> Q` par exemple).

1.2.3 Les types avec une égalité décidable

L'égalité utilisée dans le système Coq est l'égalité de Leibniz. Prouver l'égalité entre deux éléments n'est pas toujours décidable. La théorie développée dans la bibliothèque SSReflect distingue le cadre où l'on peut décider de l'égalité.

La structure d'égalité décidable est l'une des premières structures de base définies dans la bibliothèque SSReflect. On peut décider de l'égalité de deux éléments de type `T` si on a un algorithme qui prend en argument ces deux éléments et qui retourne le booléen `true` si ces éléments sont égaux, et le booléen `false` sinon. La structure est donc définie ainsi[1] :

```
Structure eqType := EqMixin {
   sort : Type ;
   eq_op : sort -> sort -> bool ;
    _ : forall x y, reflect (x = y) (eq_op x y)
  }
```

[1]Enfin presque, pour des raisons de clarté j'ai donné une définition plus simple.

La notation utilisée pour `eq_op x y` est `(x == y)`. L'expression `(x == y)` représente donc une égalité booléenne. Bien sûr, l'algorithme caché derrière la notation `==` dépend du type de `x` et de `y`, s'ils sont de type `nat`, alors le `==` représente l'algorithme qui teste l'égalité de deux éléments de type `nat`, mais s'ils sont de type `bool`, alors le `==` représente le test d'égalité de deux booléens. Le sytème COQ retrouve les bonnes instances de la notation grâce à un système de structures canoniques. Cependant dans certains cas on peut demander au système de choisir une interprétation particulière du signe `==` avec la notation `(x == y :> T)`, ici on demande au système de voir les éléments `x` et `y` comme des éléments de type `T`[1], et donc l'algorithme de comparaison utilisé sera celui instancié sur le type `T`.

1.2.4 Les types finis

Les types finis sont définis dans la bibliothèque SSREFLECT comme les types dont tous les éléments peuvent être énumérés dans une séquence. Ils sont définis dans une structure appelée `finType`. Ici nous allons présenter dans un premier temps le type fini appelé `ordinal`, car nous l'utiliserons couramment dans la suite. Nous verrons ensuite quelques notations à propos des fonctions sur un type fini.

Les ordinaux

Si `n` est un entier, alors le type `ordinal n` contient tous les entiers strictement plus petit que `n`. Il est défini dans la bibliothèque SSREFLECT comme suit :

```
Inductive ordinal (n : nat) := Ordinal m (_ : m < n).
```

Le type `ordinal n` est noté `'I_n`. Ainsi, un élément `x` de type `'I_n` est un entier muni d'une preuve qu'il est plus petit que `n`. Il y a donc une coercition naturelle du type `'I_n` dans le type `nat`, cette coercition s'appelle `nat_of_ord`.

Les fonctions sur les types finis

Si nous avons une fonction `f` de type `aT -> rT` avec `aT : finType` et `rT` un type quelconque, alors cette fonction a un nombre fini d'images, donc l'image de `f` peut être représentée par une séquence. La théorie de ces fonctions est développée dans le fichier `finfun.v` de la bibliothèque SSREFLECT. Le type de ces fonctions est noté `{ffun aT -> rT}`. Pour définir une fonction qui a ce type, on peut utiliser la notation `[ffun x => expr]` où `expr` est le corps de la fonction.

1.2.5 Les opérateurs de familles

Les opérateurs de familles sont les opérateurs que l'on utilise dans des expressions comme :

[1]Il faut donc soit que le type de `x` et de `y` soit convertible au type `T`, soit qu'il existe une coercition du type de `x` et de `y` vers le type `T`.

$$\sum_{i=n}^{p} x_i \qquad \prod_{i \in S} a_i \qquad \bigcup_{i \in S} A_i$$

Le fichier `bigop.v` de la bibliothèque SSReflect contient des définitions et des lemmes qui permettent d'exprimer formellement, et de manipuler ce genre d'expression. C'est ce que nous allons présenter brièvement ici.

Supposons que nous avons un opérateur `op` avec la notation suivante :

```
Variables (T : Type) (op : T -> T -> T).
Notation "x * y" := (op x y).
```

Les opérateurs de familles sont un moyen d'exprimer de manière concise une expression comme :

```
x1 * x2 * ... * xk
```

Autrement dit, on s'en sert pour *appliquer*, dans un certain sens, un opérateur à une famille d'éléments en itérant plusieurs fois l'opération. Il existe une fonction `foldr` définie dans la bibliothèque SSReflect, qui permet de faire cela sur les éléments d'une séquence.

La fonction `foldr` est définie comme suit :

```
Fixpoint foldr f x0 s := if s is x :: s' then f x (foldr s' f) else x0.
```

Donc dans notre exemple, nous aurons :

- `foldr op x0 [::] = x0`
- `foldr op x0 [:: x1] = x1 * x0`

Et de manière plus générale :

- `foldr op x0 [:: x1, x2, ..., xk] = x1 * (x2 * ... * xk * x0)`

L'argument `x0` de la fonction `foldr` est l'élément par défaut retourné quand la fonction est appelé sur la séquence vide. C'est pour cela que ci-dessus l'argument `x0` apparaît à la fin de chaque expression. Donc pour que la fonction `foldr` ait le comportement attendu, il suffit que `x0` soit un élément neutre à droite pour l'opérateur `op`.

Dans ce qui est fait dans le fichier `bigop.v`, la fonction `foldr` n'est pas utilisée directement sur la séquence `[:: x1, ..., xk]` mais sur la séquence des indices (i.e la séquence `[:: 1, 2, ... k]` ; et donc au lieu d'utiliser directement la fonction `op` comme premier argument, on applique d'abord la fonction `fun (i : nat) => xi` avant de faire appel à l'opérateur `op`.

Tout ce mécanisme est caché principalement derrière la notation suivante :

```
\big[op/id]_(i <- s) F i
```

où `op` est l'opérateur qui est itéré, `id` est (en général) l'élément neutre pour l'opérateur `op`, `s` la séquence des indices, et `F i` l'expression des termes auquels on *applique* l'opérateur `op`.

Par exemple, si `addn` est l'addition sur les entiers, et `U_` est une fonction sur les entiers alors :

`\big[addn/0]_(i <- [:: 2 ; 5 ; 7]) U_ i = U_ 2 + U_ 5 + U_ 7.`

Il existe plusieurs variantes de cette notation qui sont présentées dans le tableau suivant :

	Contexte	Notation générale	Exemple avec l'opérateur somme
i)		`\big[op/id]_(m <= i < n) F i`	$\sum_{i=m}^{n-1} F_i$
ii)		`\big[op/id]_(i < n) F i`	$\sum_{i<n} F_i$
iii)	`T : finType` et S est un ensemble fini qui contient tous les éléments du type `T`	`\big[op/id]_(i : T) F i`	$\sum_{i \in S} F_i$

Dans l'expression de la ligne i), nous avons `i : nat`, alors que dans l'expression de la ligne ii) nous avons `i : 'I_n`.

Dans les cas particuliers de la somme et du produit, on peut utiliser respectivement les notations `\sum_(i <- s) F i` et `\prod_(i <- s) F i`.

Il est également possible d'*appliquer* l'opérateur `op` seulement aux éléments dont les indices vérifient une certaine propriétés `P`. Pour cela on écrira :

`\big[op/id]_(i <- s | P i) F i`

Par exemple pour l'expression suivante :

$$\sum_{\substack{i<n \\ i \text{ premier}}} F_i$$

nous écrirons :

`\sum_(i < n | premier i) F i`

Le fichier `bigop.v` de la bibliothèque SSReflect contient des lemmes qui permettent de faire les manipulations usuelles sur ces opérateurs de familles. Pour plus de détails, le lecteur pourra lire [4].

1.2.6 Les structures algébriques

Les structures algébriques (groupe, anneau, corps, etc) sont définies dans le fichier `ssralg.v` de la biliothèque SSReflect. Nous allons voir ici les principales notations associées à chacune de ces strucures.

Les groupes commutatifs

La structure de goupe commutatif correspond à la structure `zmodType` de la bibliothèque SSReflect. Cette structure possède une loi commutative notée `+` avec un élément neutre noté `0` et une fonction « opposé ».

On peut voir un groupe commutatif comme un $\mathbb{Z}$-module[1] en définissant la multiplication par un entier ainsi : $x * n = \underbrace{x + \ldots + x}_{n fois}$. Cette opération est notée dans la bibliothèque SSReflect comme `x *+ n`, dans cette expression nous avons `x : T` avec `T : zmodType` et `n : nat`. L'opposé de `x *+ n` est noté `x *- n`. Intuitivement on peut lire ces deux dernières expressions respectivement comme `x` multiplié par `+n` et `x` multiplié par `-n`.

Les anneaux

La structure d'anneau correspond à la structure nommée `ringType` dans la bibliothèque SSReflect. Cette structure est basée sur la strucure `zmodType`, et possède en plus une loi multiplicative noté `*` dont l'élément neutre est noté `1`. La structure `ringType` demande, en plus des propriétés usuelles qui définissent la structure d'anneau, une preuve que `1 != 0`.

Si nous avons `R : ringType` alors la notation `n%:R`, où `n : nat`, est le plongement de l'entier `n` dans l'anneau `R`, en fait c'est une notation pour `1 *+ n`.

On note x^n par `x ^+ n`.

Les anneaux commutatifs et avec des éléments inversibles

La structure d'anneau commutatif s'appelle `comRingType`, elle est basée sur la structure `ringType`. Il existe une autre structure basée sur la structure `ringType` appelée `unitRingType`. Cette structure possède un prédicat appelé `GRing.unit` qui représente tous les éléments inversibles de l'anneau sur lequel il est basé. Le fait qu'un élément `x` est inversible s'écrit `x \is a GRing.unit`. L'inverse de `x` est noté `x ^- 1`, et de manière plus générale, l'inverse de `x ^+ n` est noté `x ^- n`.

La structure qui réunit les propriétés des deux structures ci-dessus s'appelle `comUnitRingType`.

Les autres structures

Dans la suite de la hiérarchie des structures algébriques, il y a la structure `idomainType` pour les anneaux intègres, la structure `fieldType` pour les corps, et la structure `closedFieldType` pour les corps algébriquement clos.

Les L-modules

Une dernière structure que l'on va présenter ici est la structure de L-module, appelé dans la bibliothèque SSReflect `lmodType`.

Si nous avons `R : ringType` et `V : lmodType R` alors `V` est un module, et `R` représente l'ensemble des scalaires. Si nous avons `a : R` et `v : V` alors la multiplication par un scalaire est notée `a *: v`.

[1] d'où le nom `zmodType`.

On utilisera principalement cette notation avec les polynômes et les matrices par exemple.

1.2.7 Les polynômes

Les polynômes peuvent être vus comme une suite de coefficients qui s'annule à partir d'un certain rang. Si on supprime tous les zéros de cette suite à partir du rang auquel elle s'annule, alors on obtient une séquence dont le dernier élément est non nul, c'est ainsi que sont formalisés les polynômes dans la bibliothèque SSReflect :

```
Variable R : ringType.

Record polynomial := Polynomial {
  polyseq :> seq R;
  _ : last 1 polyseq != 0}.
```

Le symbole `:>` permet de déclarer `polyseq` comme une coercition du type des polynômes vers le type des séquences. La séquence vide représentant le polynôme nul doit vérifier la définition ci-dessus. Or lorsque la fonction `last` est appliquée à la séquence vide, elle retourne un élément par défaut. Cet élément par défaut doit être différent de `0`, et le seul autre élément de l'anneau `R` dont on connait l'existence est `1`. Donc pour que cette définition de polynôme soit valide, il faut que `1 != 0`. C'est principalement pour cette raison que l'axiome `1 != 0` a été rajouté comme axiome dans la structure `ringType`.

Nous allons voir d'abord quelques généralités sur les polynômes, ensuite nous parlerons de la divisibilité sur les polynômes.

Généralité

Le type des polynômes est noté `{poly R}`, ainsi si `p : {poly R}` alors `p` est un polynôme à coefficients dans l'anneau `R`.

Voici un tableau qui résume les principales notations sur les polynômes :

Notation SSReflect	Description
`c%:P`	Le polynôme constant c (avec `c : R`)
`'X`	Le polynôme X (où X est l'indéterminée)
`lead_coef p`	Le coefficient dominant du polynôme p
`p \is monic`	Le polynôme p est unitaire
`p.[a]`	La valeur du polynôme p évalué en a (i.e $p(a)$)

Il n'y a pas vraiment de définition pour le degré d'un polynôme. On parle plutôt de la taille de la séquence des coefficients. Comme il existe une coercition des polynômes vers les séquences, on peut directement écrire `size p`. Donc pour un polynôme `p` non nul, `(size p).-1` est égal au degré de `p`.

Divisibilité et racines

Les notations liés à la divisibilité et aux racines d'un polynôme sont présentées dans le tableau suivant :

Notation SSReflect	Description
`p %/ q`	p divisé par q
`p %% q`	p modulo q
`p %| q`	p divise q
`p %= q`	`p %| q` et `q %| p`
`gcdp p q`	$\text{pgcd}(p, q)$
`root p x`	x est une racine de p
`\mu_x p`	multiplicité de x dans p

La relation `%=` est une relation d'équivalence. Si `p %= q` alors cela veut dire que l'on peut passer de l'un à l'autre en multipliant par un scalaire inversible. En mode mathématique, nous noterons cette relation $p \sim q$.

1.2.8 Les matrices

Dans la bibliothèque SSReflect, les matrices sont définies comme suit :

```
Inductive matrix := Matrix (_ : {ffun 'I_m * 'I_n -> R}).
```

Ce type inductif a un constructeur `Matrix` qui prend en argument une fonction dont toutes les valeurs correspondent aux coefficients de la matrice. Cette fonction peut être récupérée par la fonction `mx_val` définie comme suit :

```
Definition mx_val m n (A : matrix m n) :=
  match A with Matrix g => g end.
```

On peut ainsi définir la coercition suivante :

```
Definition fun_of_matrix m n A (i : 'I_m) (j : 'I_n) := mx_val A (i,j).
Coercion fun_of_matrix : matrix -> Funclass.
```

Cette coercition permet de voir les matrices comme des fonctions. Ainsi, si nous avons une matrice `A`, alors on peut écrire `A i j` le coefficient de la ligne `i` et de la colonne `j` de `A`. Le type des matrices est noté de manière générale `'M[R]_(m,n)` où `R` est le type des coefficients de la matrice et `m` et `n` ses dimensions. Si il n'y a pas d'ambiguïté sur le type des coefficients, alors on peut écrire `'M_(m,n)`. Dans le cas des matrices carrées on notera `'M[R]_n` ou `'M_n`.

Si l'on connait l'expression générale des coefficients d'une matrice, alors on peut la définir avec la notation suivante :

```
\matrix_(i < m , j < n) E i j
```

où `E` est l'expression générale des coefficients. Le type de la matrice ainsi définie est `'M_(m,n)`. Dans le cas des matrices carrées, pour la taille de la matrice, on peut juste préciser `(i, j < n)` ; et si il n'y a pas d'ambiguïté sur la taille de la matrice alors on précisera simplement `(i,j)`.

Par exemple la matrice définie comme suit :

```
\matrix_(i, j < 3) 2 *+ (i == j) + (i.+1 == j)
```

correspond à la matrice :

$$\begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

Comme il y a une coercition du type `bool` vers le type `nat`, l'expression `(i == j)` sera interprétée comme `0` ou `1`, et elle ne vaudra `1` que dans le cas où `i` et `j` sont égaux ; c'est la raison pour laquelle les 2 apparaissent sur la diagonale. En fait `(i == j)` est une façon d'écrire δ_{ij}, le symbole de Kronecker, et `a *+ (i == j)` est une façon d'écrire $a.\delta_{ij}$.

Ici nous avons montré l'exemple d'une matrice carrée où `i` et `j` ont le même type. Mais dans le cas d'une matrice rectangulaire, `i` et `j` ont des types différents, et donc l'expression `(i == j)` est mal typée. Par exemple, si nous avions écrit :

```
\matrix_(i < 2, j < 3) 2 *+ (i == j) + (i.+1 == j)
```

Nous aurions `i : 'I_2` et `j : 'I_3`. Pour que la définition soit correcte il faut préciser au système Coq que l'on veut que la comparaison entre `i` et `j` se fasse sur les entiers. La définition correcte est donc :

```
\matrix_(i < 2, j < 3) 2 *+ (i == j :> nat) + (i.+1 == j)
```

Pour finir voici les notations usuelles sur les matrices :

Notation SSReflect	Description
`A i j`	Le coefficient A_{ij} de la matrice A
`a%:M`	La matrice qui ne contient que des a sur la diagonale (par exemple `1%:M` est la matrice identité)
`A *m B`	$A * B$
`\det A`	Le déterminant de A
`A \in unitmx`	A est inversible

1.2.9 Matrices et polynômes

Dans cette thèse nous serons amenés à évoquer les notions de polynôme caractéristique d'une matrice et de polynôme minimal. Ces notions sont formalisées dans la bibliothèque SSReflect dans le fichier `mxpoly.v`.

Soit $\mathbb{A}$ un anneau, $p \in \mathbb{A}[X]$ et $A \in \mathcal{M}_n(\mathbb{A})$. Alors $p(A)$ est la matrice qui correspond à l'évaluation du polynôme p en la matrice A. Par exemple, si nous

avons $p = 2X^2 - 3X + 4$ alors $p(A) = 2A^2 - 3A + 4I$. Si p est tel que $p(A) = 0$ alors on dit que p est un polynôme annulateur de la matrice A.

La matrice $XI - A$ est la matrice caractéristique de A. Le déterminant de cette matrice est un polynôme appelé polynôme caractéristique de la matrice A. Ce polynôme est un polynôme annulateur de la matrice.

Le polynôme unitaire et annulateur de la matrice A de plus petit degré est appelé polynôme minimal de la matrice A. Une propriété caractéristique du polynôme minimal est que c'est un polynôme annulateur qui divise tout les autres polynômes annulateurs de la matrice.

Le tableau ci-dessous donne le nom des fonctions définies dans la bibliothèque SSReflect qui correspondent à ce que l'on a décrit ci-dessus :

Notation SSReflect	Description
`horner_mx A p`	$p(A)$
`char_poly_mx A`	$XI - A$
`char_poly A`	polynôme caractéristique de A
`mxminpoly A`	polynôme minimal de A

1.2.10 Les structures ordonnées

Les structures telles que les corps réels clos, ou les corps archimédiens sont définis dans le fichier `ssrnum.v` de la bibliothèque SSReflect. La structure ordonnée la plus générale est basée sur la structure d'anneau commutatif intègre (`idomainType`), et contient en plus une comparaison booléenne large notée `<=`, une comparaison booléenne stricte notée `<`, et une fonction valeur absolue ou module notée `` `|.| ``. Cette structure est appelée `numDomainType`. Sur cette structure, les fonctions `maxr` et `minr` désignent respectivement les fonctions maximum et minimum.

Voici les structures qui sont implémentées dans le fichier `ssrnum.v` :

Nom de la structure	Description
`numDomainType`	c'est la structure décrite plus haut
`numFieldType`	`numDomainType`, mais basé sur un corps
`numClosedFieldType`	`numDomainType` basé sur un corps algébriquement clos
`realDomainType`	`numDomainType` avec un ordre total
`realFieldType`	Corps réel
`archiFieldType`	Corps archimédien
`rcfType`	Corps réel clos

Ces structures (ainsi que la construction de nombres complexes ci-dessous) sont expliquées plus en détails dans la thèse de Cyril Cohen [8].

1.2.11 Les nombres complexes

Les nombres complexes sont définis comme étant une paire de nombres :

```
Inductive complex (R : Type) : Type := Complex {Re : R; Im : R}.
```

On utilisera en général la notation suivante :

```
Local Notation C := (complex R).
```

La notation `x%:C` représente un réel plongé dans les complexes, c'est une notation pour `Complex x 0`.

On peut munir le type `complex R` des structures ordonnées `numDomainType` et `numFieldType` si `R` est de type `rcfType` (i.e. c'est un corps réel clos). En effet, pour définir le module d'un nombre complexe, il faut définir la fonction « racine carrée ». Or la racine carrée d'un nombre a est définie comme une racine du polynôme $X^2 - a$, donc pour que cette fonction soit bien définie, il faut[1] que `R` soit un corps réel clos.

L'ordre partiel défini sur les complexes est le suivant :

```
Definition lec x y := (Im x == Im y) && (Re x <= Re y).
Definition ltc x y := (Im x == Im y) && (Re x < Re y).
```

La structure de `numDomainType` est instanciée sur le type `complex` avec les deux définitions ci-dessus pour l'ordre large et l'ordre strict, et avec la fonction “module” suivante :

```
Definition normc x := (Num.sqrt ((Re x)^+2 + (Im x)^+2))%:C.
```

Pour pouvoir déclarer les nombres complexes comme étant une instance de la structure `numDomainType`, on doit plonger le module d'un nombre complexe dans `C`. Car si `T` est le type principal sur lequel est défini la structure `numDomainType`, alors le type de la fonction `` `|.| `` est `T -> T`. Or, dans notre cas, `T` est instancié par `C`, donc la fonction module doit être de type `C -> C`. Ceci a pour inconvénient que le module d'un nombre complexe est un nombre complexe.

1.3 Conclusion

Ce chapitre explique les principaux outils utilisés dans le travail de formalisation. Il présente aussi les notations usuelles qui seront utilisées dans la suite de cette thèse.

[1] Enfin pas tout à fait. Disons que la structure `rcfType` est la plus petite structure implémentée dans la bibliothèque SSReflect telle que la fonction « racine carrée »soit bien définie.

Chapitre 2

Forme de Jordan d'une matrice

Un résultat important en algèbre linéaire est le fait qu'une matrice soit semblable à sa forme normale de Jordan. La formalisation de ce résultat passe d'abord par la formalisation de nombreux concepts comme les matrices diagonales par blocs, les matrices compagnes, les facteurs invariants, et les matrices semblables ou équivalentes. Ce résultat utilise également d'autres formes normales de matrices comme la forme normale de Smith et la forme normale de Frobenius d'une matrice.

Les travaux présentés dans ce chapitre ont été menés en commun avec Maxime Dénès (voir [7]), et ont été inspirés de la thèse de Patrick Ozello [37] et de celle de Isabelle Gil [16].

Nous verrons dans une première section comment les matrices diagonales par blocs ont été formalisées. Dans la section suivante, nous parlerons de la similitude et de l'équivalence entre deux matrices. Ensuite nous discuterons sur la forme normale de Smith, sur ses propriétés, et montrerons quelques lemmes importants qui seront utiles par la suite. Dans une quatrième section, nous donnerons les définitions formelles de matrices compagnes et des facteurs invariants ; nous pourrons ainsi définir la forme normale de Frobenius d'une matrice. Enfin nous montrerons comment nous obtenons la forme normale de Jordan d'une matrice.

2.1 Matrices diagonales par blocs

Une matrice diagonale par blocs est une matrice de la forme :

$$\begin{pmatrix} A_1 & 0 & \dots & 0 \\ 0 & A_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & A_k \end{pmatrix}$$

où les A_i sont des matrices carrées quelconques.

Ces matrices seront utilisées pour exprimer les formes normales de Frobenius et de Jordan d'une matrice.

Ce qui est intéressant dans les matrices diagonales par blocs est qu'en général si l'on veut faire des opérations (ou montrer des propriétés), on peut se ramener à les faire (ou les montrer) sur chaque bloc ; ce qui permet dans certains cas de se ramener à un problème plus simple.

Certains choix de formalisation dans la bibliothèque SSREFLECT font que la définition formelle de matrice diagonale par blocs n'est pas aussi simple qu'on le voudrait. Nous expliquerons donc dans un premier temps quelle définition de matrice diagonale par blocs nous voudrions avoir, et ensuite nous verrons quels problèmes nous avons rencontrés avec cette définition, et comment ils ont été résolus.

2.1.1 Définition idéale

Dans la bibliothèque SSREFLECT, il existe une fonction qui s'appelle `block_mx`. Cette fonction permet de construire une matrice à partir de blocs matriciels. Par exemple, si `A`, `B`, `C` et `D` sont des matrices (avec des dimensions adéquates) alors `block_mx A B C D` est la matrice

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Si `B` et `C` sont des matrices nulles, et si `A` et `D` sont des matrices carrées, alors `block_mx A 0 0 D` est une matrice diagonale par blocs avec comme blocs diagonaux les matrices `A` et `D`. L'idée de base pour construire les matrices diagonales par blocs est d'appeler récursivement la fonction `block_mx`.

À chaque appel de la fonction `block_mx`, il faudra lui donner en arguments successivement les blocs diagonaux de la matrice que l'on veut construire. Ces blocs peuvent être de tailles différentes, et donc de types différents. Une première difficulté va être d'exprimer l'ensemble de ces blocs diagonaux.

Une solution pourrait être d'utiliser le type des paires dépendantes de COQ. Dans la bibiliothèque standard de COQ, il y a un type `sigT` défini comme suit :

```
Inductive sigT (A : Type) (P : A -> Type) : Type :=
  existT : forall x : A, P x -> sigT P.

Notation "{ x : A & P }" := (sigT (fun x : A => P))
```

Dans notre cas, si `A` est le type `nat` et `P` est la fonction `fun n => 'M_n`, alors `{n : nat & 'M_n}` est le type des paires composées d'un entier et d'une matrice. Ces paires sont dépendantes dans le sens où le type de la matrice dépend de l'entier. De plus, pour tout `k : nat` et pour toute matrice `M : 'M_k` l'expression `existT k M` est de type `{n : nat & 'M_n}`. Ainsi une sequence `s` dont le type est `seq {n : nat & 'M_n}` peut contenir des matrices de types différents (comme deuxième élément d'une paire), nous pouvons donc nous en servir pour parler des blocs diagonaux. Ce qui nous donnerait comme première définition :

```
Fixpoint diag_block_mx (s : seq {n : nat & 'M[R]_n}) :=
 match s return 'M[R]_(size_sum s) with
   | [::]=> 0
   | (existT k A) :: s'=> block_mx A 0 0 (diag_block_mx s')
 end.
```

où `size_sum s` est la fonction qui donne la taille de la matrice diagonale par blocs, elle consiste simplement à faire la somme des tailles de chaque bloc.

Cette définition est suffisante pour prouver qu'une matrice est semblable à sa forme de Jordan, car la seule chose que l'on fait avec les matrices diagonales par blocs est de prouver leur similitude ou leur équivalence avec d'autres matrices. Par contre, il est difficile de définir les opérateurs usuels sur ces matrices.

Nous allons rencontrer un premier problème lorsque nous voudrons exprimer la somme de deux matrices diagonales par blocs. La matrice `diag_block_mx s1` et la matrice `diag_block_mx s2`, pour des séquences `s1` et `s2` différentes, ont respectivement comme taille `size_sum s1` et `size_sum s2` ; autrement dit elles ont des types différents (même si l'on peut prouver que `size_sum s1 = size_sum s2`). En particulier on ne pourra même pas écrire l'expression suivante :

```
diag_block_mx s1 + diag_block_mx s2.
```

Il est possible de contourner ce problème en utilisant la fonction `castmx` de la bibliothèque SSReflect. Si nous avons deux matrices `A` et `B`, respectivement de type `'M_(m1,n1)` et `'M_(m2,n2)`, et deux preuves d'égalités `eqm : m1 = m2` et `eqn : n1 = n2` alors la matrice `cast_mx (eqm,eqn) B` a le même type que la matrice `A` (i.e `'M_(m1,n1)`). Donc si `eq_size` est une preuve d'égalité entre `size_sum s1` et `size_sum s2` alors nous pouvons écrire l'addition de deux matrices diagonales par bloc comme suit :

```
diag_block_mx s1 + castmx (eq_size,eq_size) diag_block_mx s2
```

Malgré l'utilisation de la fonction `castmx`, nous nous heurterons à un deuxième problème pour exprimer qu'additionner deux matrices diagonales par blocs revient à les additionner bloc à bloc. Pour cela, il faudrait montrer que chacun des blocs des séquences `s1` et `s2` sont deux à deux de la même taille, et à partir de cette preuve obtenir les preuves d'égalité qui permetteront à l'aide de la fonction `castmx` d'écrire les sommes des blocs deux à deux. Cette définition de matrice diagonale par blocs n'est donc pas adaptée pour exprimer ce genre de propriété.

Si nous voulons additionner deux matrices diagonales par blocs, c'est qu'elles ont, en général, le même découpage en blocs, car cela permet de faire l'addition bloc à bloc. Si elles n'ont pas le même découpage en blocs, alors l'information que les matrices sont diagonales par blocs n'apporte rien, et l'on doit faire l'addition comme sur deux matrices quelconques. Il est donc intéressant que deux matrices diagonales par blocs, ayant le même découpage en blocs, aient le même type.

Dans la description précédente, nous avons vu que la dimension d'une matrice diagonale par blocs est donné par la fonction `size_sum`. Cette fonction prend en argument une liste de paires dépendantes tailles/matrices, en extrait les dimen-

sions des matrices et en fait la somme. Dans la nouvelle définition de matrice diagonale par blocs, nous allons plutôt donner comme premier argument directement une séquence d'entiers qui correspondra à la séquence des tailles des blocs ; autrement dit cette séquence donnera le découpage en blocs de la matrice.

Maintenant, pour exprimer les blocs de la matrice, nous ne pouvons plus utiliser une séquence de paires dépendantes. Car sinon il faudrait s'assurer que les tailles données comme premier élément de chaque paire soient les mêmes que ceux de la séquence de découpage des blocs, ce que l'on veut éviter. Pour exprimer les blocs, nous allons donc utiliser une fonction. Cette fonction doit retourner comme valeur des matrices de différentes tailles, et donc de types différents. C'est pourquoi le premier argument de la fonction sera la taille de la matrice qu'elle va retourner ; pour le deuxième argument nous prendrons simplement le numéro du bloc dans la matrice diagonale par blocs. Le type de cette fonction est donc `forall n : nat, nat -> 'M_n`.

Notre nouvelle définition de matrices diagonales par blocs se présente donc ainsi :

```
Fixpoint diag_block_mx s (F : forall n : nat, nat -> 'M_n) :=
  match s return 'M_(sumn s) with
   | [::] => 0
   | n :: s' => block_mx (F n 0) 0 0
                 (diag_block_mx s' (fun n i => F n i.+1))
  end.
```

où `sumn` prend en argument une séquence d'entiers et retourne la somme de ces entiers. Voyons comment cela fonctionne sur un exemple :

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 5 & 1 \\ 0 & 0 & 0 & 9 & 6 & 6 \\ 0 & 0 & 0 & 0 & 4 & 2 \end{pmatrix}$$

Le découpage en blocs de la matrice ci-dessus correspond à la séquence de découpage `[:: 2 ; 1 ; 3]`[1]. Le bloc numéro 0 est de taille 2, il correspond donc au bloc `F 2 0` où `F` est la fonction qui retourne les blocs diagonaux de la matrice ci-dessus, c'est-à-dire n'importe quelle fonction qui vérifie les trois conditions suivantes :

$$\texttt{F 2 0} = \begin{pmatrix} 1 & 1 \\ 2 & 4 \end{pmatrix}; \qquad \texttt{F 1 1} = \begin{pmatrix} 4 \end{pmatrix}; \qquad \texttt{F 3 2} = \begin{pmatrix} 4 & 5 & 1 \\ 9 & 6 & 6 \\ 0 & 4 & 2 \end{pmatrix}$$

[1] Il y a quatre façon de découper cette matrice en blocs qui correspondent aux séquences suivantes : `[:: 6]`, `[:: 2 ; 4]`, `[:: 3 ; 3]`, `[:: 2 ; 1 ; 3]`.

Comme les autres valeurs de la fonction `F` ne sont pas appelées par la fonction `diag_block_mx`, elles n'ont pas d'importance.

Avec cette définition nous pouvons exprimer facilement que la somme de deux matrices diagonales par blocs est la matrice diagonale par blocs où chaque bloc est la somme des blocs des deux premières matrices :

```
Lemma add_diag_block s F1 F2 :
 diag_block_mx s F1 + diag_block_mx s F2 =
 diag_block_mx s (fun n i => F1 n i + F2 n i).
```

Cette deuxième définition est assez proche de la définition idéale que l'on voudrait pour les matrices diagonales par blocs, car elle est proche de l'intuition que l'on s'en fait (appel récursif de la fonction `block_mx`), et permet de manipuler les matrices diagonales par blocs assez simplement. Cependant, à cause de certains choix de formalisation dans la bibliothèque SSReflect, nous allons rencontrer d'autres problèmes qui font que nous allons devoir modifier cette définition. C'est ce que nous allons voir dans la sous-section suivante.

2.1.2 Problèmes en dimension nulle

La définition de structure d'anneaux dans la bibliothèque SSReflect demande une preuve que 1 (l'élément neutre de la loi multiplicative) soit différent de 0 (l'élément neutre de la loi additive). Ceci a pour conséquence que l'ensemble des matrices carrées d'une taille fixée forme un anneau (dans le sens de SSReflect) seulement si ces matrices sont non vides, c'est-à-dire de taille non nulle (car toutes les matrices vides sont égales). Autrement dit, l'instance canonique de la structure d'anneau sur les matrices est implémentée seulement sur celles ayant un type de la forme `'M_n.+1`. Ainsi, sur les matrices n'ayant pas un type de cette forme, on ne peut pas utiliser les fonctions définies à partir des opérations génériques d'anneaux (par exempe les fonctions qui calculent la puissance, le polynôme minimal où la valeur d'un polynôme en une matrice).

Or le type des matrices diagonales par blocs définies plus haut est de la forme `'M_(sumn s)`. Nous ne pourrons donc pas utiliser sur ces matrices certaines des fonctions citées ci-dessus. En effet, si l'on essayait d'appliquer une de ces fonctions, le système Coq attendrait comme argument une structure d'anneau sur les matrices, comme cette structure est déclarée pour les matrices dont le type est de la forme `'M_n.+1`, le système va essayer de résoudre le problème d'unification `?n.+1 = sumn s` et il va échouer. Nous allons donc voir ici comment la définition de matrice diagonale par blocs a été modifiée pour que le type de ces matrices soit de la bonne forme.

2.1.3 Définition finale

Une propriété intéressante des matrices diagonales par blocs est que faire certaines opérations sur une matrice diagonale par blocs revient à faire ces opérations sur chaque bloc. Si ces opérations utilisent des opérations génériques d'anneaux,

il faut alors que chacun des blocs ait un type de la forme `'M_n.+1`. Le type de la fonction `F` de la définition de matrice diagonale par blocs devient donc `forall n, nat -> 'M_n.+1`.

Maintenant nous voulons trouver une fonction, appelons-la `size_sum`, telle que le type d'une matrice diagonale par blocs soit `'M_(size_sum s).+1`. Nous remarquons d'abord que si la séquence donnée en premier argument de la fonction `diag_block_mx` est vide, alors la matrice diagonale par blocs retournée est vide, et donc sa taille ne peut pas être de la forme `n.+1`. Il va donc falloir modifier la définition de `diag_block_mx` pour qu'elle ne renvoie jamais une matrice vide. Pour cela nous allons définir une première fonction pour laquelle nous allons séparer la séquence de découpage des blocs en deux ; c'est-à-dire qu'elle va prendre en argument un entier qui va représenter la taille du premier bloc, et une séquence qui contiendra la taille des autres blocs (à un près à cause du décalage dû au `n.+1`) :

```
Fixpoint diag_block_mx_rec k s (F : forall n, nat -> 'M_n.+1) :=
  match s return 'M_((size_sum_rec k s).+1) with
   | [::] => F k 0
   | n :: s' => block_mx (F k 0) 0 0
                (diag_block_mx_rec n s' (fun n i => F n i.+1))
 end.
```

où `size_sum_rec` est défini de telle façon que le type de `diag_block_mx_rec` ait la bonne forme :

```
Fixpoint size_sum_rec k s :=
  match s with
   | [::] => k
   | n :: s' => k + (size_sum_rec n s').+1
  end.
```

donc si `s` est vide alors `(size_sum_rec k s).+1 = k.+1` ce qui est bien le type de `F k 0`, et si `s = n :: s'` alors :

```
(size_sum_rec k s).+1 = (k + (size_sum_rec n s').+1).+1
```

et la dimension de la matrice dans la deuxième branche de la définition de `diag_block_mx_rec` est `k.+1 + (size_sum_rec n s').+1`, et comme l'addition est définie récursivement par rapport à la première variable, les deux expressions sont convertibles.

Mais pour utiliser cette définition, il faut donner en argument un entier et une séquence d'entiers, ce qui peut souvent amener à "couper" en deux une séquence pour donner les deux arguments. Pour éviter cela nous définissons la fonction `diag_block_mx` de façon à ce qu'elle prenne en argument seulement une séquence :

```
Definition size_sum s :=
  match s with
   | [::] => 0
   | n :: s' => size_sum_rec n s'
  end.
```

```
Definition diag_block_mx s (F : forall n, nat -> 'M_n.+1) :=
  match s return 'M_((size_sum s).+1) with
   | [::] => 0 (* de type 'M_1 *)
   | n :: s' => diag_block_mx_rec n s' F
 end.
```

Cette définition retourne une matrice de taille un si la séquence donnée en argument est vide. Dans la pratique cela ne devrait pas trop poser de problèmes car lorsque l'on parle d'une matrice diagonale par blocs, c'est rarement à propos d'une matrice vide. Ceci a pour conséquences que dans les énoncés des théorèmes, il faut dans certains cas rajouter l'hypothèse que la séquence des tailles des blocs soit non vide ; et dans les preuves, il faut en général faire un traitement par cas sur cette séquence pour éliminer le cas où elle est vide. Dans le cas où la séquence n'est pas vide, la fonction `diag_block_mx` a le comportement attendu.

Pour conclure cette section, cette dernière définition de matrice diagonale par blocs a l'inconvénient mineur de ne pas avoir le bon comportement dans le cas où la séquence des tailles des blocs est vide. Mais maintenant le type d'une matrice retournée par la fonction `diag_block_mx` est de la bonne forme pour que l'on puisse utiliser les opérations génériques d'anneaux dessus, ainsi que sur chacun des blocs de la matrice.

2.2 Matrices semblables et équivalentes

Rappelons que nous voulons montrer que la forme normale de Jordan d'une matrice est semblable à celle-ci. Pour cela nous utiliserons en plus de la notion de matrices semblables, celle de matrices équivalentes. Nous présenterons donc, dans cette section, la formalisation de ces deux notions, et nous verrons comment elles sont liées.

2.2.1 Définitions

Deux matrices A et B sont équivalentes s'il existe deux matrices inversibles M et N telles que $MAN = B$, et elles sont semblables s'il existe une matrice inversible P telle que $PAP^{-1} = B$. Pour la définition de matrices semblables, nous préférerons la version avec $PA = BP$, car dans la bibiliothèque SSReflect, la notation `x^-1` est définie sur la structure appelée `unitRingType`[1], or cette structure est au-dessus de la structure d'anneaux de SSReflect, et donc comme nous l'avons vu dans la section précédente, nous pourrons utiliser cette notation uniquement sur les matrices dont le type est de la forme `'M_n.+1`.

Une première définition formelle que l'on pourrait donner pour la similitude est la suivante[2] :

[1] voir section 1.2.6

[2] voir 1.2.8 pour les notations.

```
Definition similar n (A B : 'M[R]_n) :=
  (exists P : 'M_m , P \in unitmx /\ P *m A = B *m P).
```

Mais cette définition n'est pas assez souple pour être utilisée facilement. En effet, pour pouvoir écrire l'expression `similar A B`, il faut absolument que les matrices `A` et `B` aient le même type. Or nous avons signalé plus haut que la forme normale de Jordan sera exprimée comme une matrice diagonale par blocs. Donc si `A : 'M_n` alors sa forme normale de Jordan sera de type `'M_(size_sum s).+1` pour une certaine séquence `s`. En particulier, nous ne pouvons pas écrire l'énoncé qui nous intéresse, à savoir qu'une matrice est semblable à sa forme normale de Jordan ; sauf si nous décidons d'utiliser la fonction `castmx`, mais cela alourdirait les énoncés et compliquerait les preuves.

Pour éviter ces problèmes, nous définissons le prédicat `similar` pour deux matrices quelconques (pas forcément de même type). Cela implique que le prédicat doit vérifier l'égalité des tailles des matrices :

```
Definition similar m n (A : 'M[R]_m) (B : 'M[R]_n) := m = n /\
  (exists P : 'M_m , P \in unitmx /\ P *m A = (conform_mx P B) *m P).
```

Nous voyons aussi apparaître dans la définition la fonction `conform_mx`. Cette apparition est due au fait que, comme les matrices `P` et `B` n'ont pas le même type, l'expression `B *m P` est mal typée.

La fonction `conform_mx` est définie dans la bibliothèque SSReflect. Si `M` et `N` sont deux matrices alors `conform_mx M N` a le même type que la matrice `M`. Si l'on peut prouver que les matrices `M` et `N` ont des tailles différentes, alors `conform_mx M N = M`. Si les matrices `M` et `N` ont le même type alors nous avons `conform_mx M N = N`.

Nous avons donc utilisé cette fonction pour remplacer l'expression `B *m P`, qui est mal typée, par l'expression `(conform_mx P B) *m P` qui elle est bien typée. Ainsi si les matrices `A` et `B` ont la même dimension alors `(conform_mx P B)` représentera la matrice que l'on attend, sinon les matrices `A` et `B` n'ont aucune chance d'être semblables, et donc la matrice retournée par `(conform_mx P B)` importe peu, puisque l'égalité des tailles est vérifiée par ailleurs.

Pour les mêmes raisons, nous définissons de manière analogue l'équivalence entre deux matrices :

```
Variables m1 n1 m2 n2 : nat.
Definition equivalent (A : 'M[R]_(m1,n1)) (B : 'M[R]_(m2,n2)) :=
  [/\ m1 = m2, n1 = n2 & exists M, exists N,
  [/\ M \in unitmx , N \in unitmx & M *m A *m N = conform_mx A B]].
```

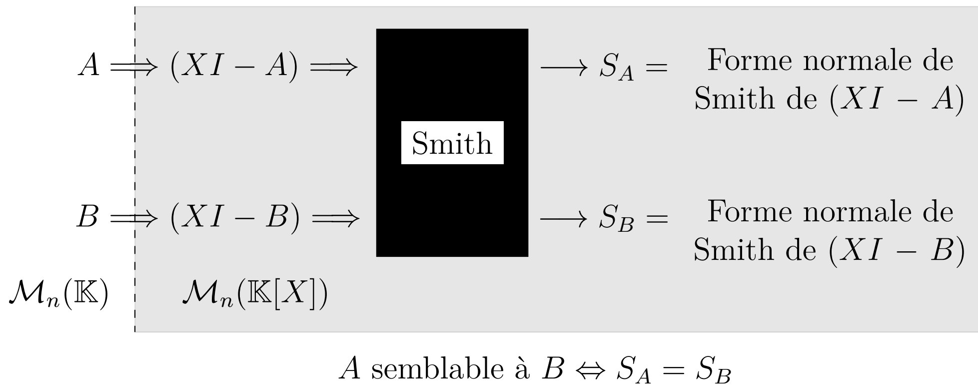
2.2.2 Théorème fondamental

Le lien entre ces deux notions d'équivalence et de similitude est donné par le théorème fondamental de similitude sur un corps :

Théorème. *Soit $\mathbb{K}$ un corps, soit A et B deux matrices de $\mathcal{M}_n(\mathbb{K})$ alors :*

$$A \text{ et } B \text{ sont semblables} \Leftrightarrow (XI - A) \text{ et } (XI - B) \text{ sont équivalentes}$$

Les matrices $(XI - A)$ et $(XI - B)$ sont les matrices caractéristiques de A et de B. Ce sont des matrices à coefficients dans l'anneau principal $\mathbb{K}[X]$. Or sur les anneaux principaux, chaque matrice a une forme normale équivalente appelée forme normale de Smith. Le théorème fondamental permet de donner une procédure simple pour tester si deux matrices sont semblables. Il suffit de prendre les matrices caractéristiques de ces matrices, de calculer leur forme normale de Smith, et de comparer les résultats.



A semblable à $B \Leftrightarrow S_A = S_B$

Ce théorème sera souvent utilisé dans la suite de cette formalisation, il permet de montrer que deux matrices sont semblables sans avoir à donner la matrice de passage entre ces deux matrices.

L'algorithme qui donne la forme normale de Smith d'une matrice et le théorème fondamental ont été respectivement implémenté et prouvé formellement par Maxime Dénès [12, 9]. Ce travail sur la forme normale de Smith a permis d'avoir une formalisation complète (sans axiomes) des formes normales de Frobenius et de Jordan d'une matrice.

Dans les sections suivantes nous allons d'abord présenter ce qu'est la forme normale de Smith d'une matrice et ensuite nous verrons le cheminement pour passer de la forme normale de Smith à celle de Jordan.

2.3 Forme normale de Smith

Nous expliquons dans cette section ce qu'est la forme normale de Smith d'une matrice. Nous voyons ensuite comment l'algorithme qui donne cette forme normale est spécifié formellement, et montrons quelques propriétés importantes.

2.3.1 Spécification

La forme normale de Smith d'une matrice A est une matrice équivalente à A qui se présente de la manière suivante :

$$\begin{pmatrix} d_1 & 0 & \dots & \dots & & \dots & 0 \\ 0 & d_2 & \ddots & & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & & \vdots \\ 0 & \dots & 0 & d_k & 0 & \dots & 0 \end{pmatrix}$$

avec la particularité que $\forall i, 1 \leq i < k, d_i \mid d_{i+1}$. Avoir un pgcd est suffisant pour que la forme normale de Smith existe. Ce qui est le cas pour les polynômes à coefficients dans un corps, qui est le contexte dans lequel nous utiliserons la forme normale de Smith.

Pour exprimer cette matrice, nous définissons la fonction `diag_mx_seq` qui prend en argument une séquence et qui retourne une matrice diagonale dont les éléments diagonaux sont les éléments de la séquence[1] :

```
Variable R : ringType.

Definition diag_mx_seq m n (s : seq R) : 'M[R]_(m,n) :=
  \matrix_(i < m, j < n) s'_i *+ (i == j :> nat).
```

Les dimensions de la matrice retournée par la fonction `diag_mx_seq` sont ses deux premiers arguments. Ceci permet une utilisation très souple de cette fonction. En effet, si nous avons besoin dans une expression que la matrice retournée par `diag_mx_seq` soit d'un type particulier, il suffit de donner les premiers arguments adéquats. Mais cette souplesse implique qu'il faut ajouter certaines contraintes dans les énoncés des propriétés de la fonction `diag_mx_seq`, pour établir un lien entre les dimensions de la matrice retournée par la fonction et la séquence donnée en argument.

Par exemple, pour la sequence `s = [:: 1 ; 2 ; 3]` nous avons :

`diag_mx_seq 4 2 s =`
$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix};$$

`diag_mx_seq 5 4 s =`
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix};$$

`diag_mx_seq 3 3 s =`
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Nous voyons ici que selon les valeurs de `m` et de `n` la matrice `diag_mx_seq m n s` n'est pas toujours la matrice à laquelle on pense. En fait, pour que la matrice `diag_mx_seq m n s` soit une matrice telle que sur sa diagonale on puisse lire toute la séquence `s`, et seulement la séquence `s`, il faut que la taille de la séquence soit égale au minimum de `m` et de `n`.

[1] voir 1.2.8 pour les notations.

Une fonction `Smith` nous est fournie par [9] pour calculer la forme normale de Smith d'une matrice `A : 'M_(m,n)`. Cette fonction retourne un triplet `(M,d,N)`, où `M` et `N` sont des matrices et `d` une séquence qui vérifient les trois spécifications suivantes :

- `M *m A *m N = diag_mx_seq m n d`
- `sorted %| d` (La séquence `d` est triée pour la relation de division sur l'anneau `R`)
- `M \in unitmx & N \in unitmx` (Les matrices `M` et `N` `sont inversible`)

Nous pouvons fusionner le premier et le dernier point pour avoir les deux spécifications suivantes :

- `equivalent A (diag_mx_seq m n d)`
- `sorted %| d`

Nous allons montrer dans la section suivante qu'une séquence vérifiant les spécifications ci-dessus est unique modulo une relation que nous définirons.

2.3.2 Unicité

Le calcul de la forme normale de Smith utilise des calculs de pgcd sur les coefficients de la matrice. Le pgcd de deux éléments est unique modulo la relation d'équivalence $\sim$ définie de la manière suivante :

$$a \sim b \Leftrightarrow a|b \text{ et } b|a$$

La forme normale de Smith est donc unique modulo cette relation d'équivalence sur les coefficients.

Nous allons montrer ici un résultat important sur la forme normale de Smith d'une matrice qui permet d'en déduire l'unicité, mais qui sera aussi très utile dans la suite de la formalisation.

Théorème 1. *Soit A une matrice à coefficients dans un anneau principal. Soit d une séquence vérifiant les spécifications suivantes :*

- *La séquence d est triée pour la relation de division.*
- *La matrice diagonale dont les éléments diagonaux sont les éléments de d est équivalente à la matrice A.*

Si l'on note $\wedge$ le pgcd, si d_i désigne le ième élément de la séquence d et si $|A|_k$ désigne l'ensemble des mineurs d'ordre k de la matrice A, alors nous avons l'identité suivante :

$$\prod_{i=1}^{k} d_i \sim \bigwedge_{x \in |A|_k} x$$

Un mineur d'une matrice A est le déterminant d'une sous-matrice de A.

Avant de montrer ce résultat, nous allons voir sur un exemple comment on utilise ce théorème pour déterminer de façon "unique" la forme normale de Smith d'une matrice.

Exemple 1. Prenons la matrice dans $\mathcal{M}_3(\mathbb{Z}[X])$ suivante :

$$M = \begin{pmatrix} 5 & 2X & X+3 \\ -X+6 & X+1 & 4 \\ X+9 & 4X & 3X+5 \end{pmatrix}$$

Le théorème pour $k = 1$ dit que d_1 est égal au pgcd de tous les mineurs d'ordre 1 de la matrice M. Les mineurs d'ordre 1 de la matrice M sont les déterminants des sous-matrices de tailles 1 de M, i.e ce sont les coefficients de la matrice M. Ici on voit que les polynômes M_{11} et M_{23} sont premiers entre eux, donc le pgcd des coefficients de la matrice est est 1. Nous avons donc $d_1 = 1$.

Remarque 1. *En fait nous avons plus exactement $d_1 \sim 1$, donc d_1 peut être -1 ou 1. Si les coefficients de la matrice M avaient été dans $\mathbb{Q}[X]$ au lieu de $\mathbb{Z}[X]$ alors d_1 aurait pu être n'importe quel rationnel non nul. Quelque soit le choix de d_1, la matrice diagonale obtenue à la fin respecte les spécifications de l'algorithme de la forme normale de Smith. Cette remarque est valable pour tous les d_i.*

Pour $k = 2$, nous avons $d_1 * d_2$ qui est égal au pgcd de tous les mineurs d'ordre 2 de la matrice M. Dans la suite nous noterons $|M|_{IJ}$ le déterminant de la sous matrice de M définie par les lignes qui sont dans I et par les colonnes qui sont dans J. Par exemple nous avons :

$$|M|_{\{1,2\}\{1,2\}} = \begin{vmatrix} 5 & 2X \\ -X+6 & X+1 \end{vmatrix} = 2X^2 - 7X + 5 = (X-1)(2X-5)$$

De même on a :

$$|M|_{\{1,3\}\{1,3\}} = \begin{vmatrix} 5 & 3+X \\ X+9 & 3X+5 \end{vmatrix} = -X^2 + 3X - 2 = (X-1)(-X+2)$$

$$|M|_{\{1,2\}\{2,3\}} = \begin{vmatrix} 2X & 3+X \\ X+1 & 4 \end{vmatrix} = -X^2 + 4X - 3 = (X-1)(-X+3)$$

$$\vdots$$

Si on continue, on s'aperçoit que $(X-1)$ est le seul facteur commun à tous les mineurs d'ordre 2 de M, donc $d_1 * d_2 = d_2 = (X-1)$.

Enfin pour $k = 3$, nous avons $d_1 * d_2 * d_3$ qui est égal au pgcd des mineurs d'ordre 3 de la matrice M. Or la seule sous-matrice de M d'ordre 3 est la matrice

M elle même. Donc $d_1 * d_2 * d_3 = \det M$ ce qui donne après calcul $(X-1) * d_3 = (X-1)^2(X-2)$ soit $d_3 = (X-1)(X-2)$. Ainsi la forme normale de Smith de M est la matrice :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & (X-1) & 0 \\ 0 & 0 & (X-1)(X-2) \end{pmatrix}$$

□

Nous avons vu comment utiliser le théorème précédent pour déterminer de manière unique (modulo la relation $\sim$) la forme normale de Smtih d'une matrice. Maintenant nous allons voir comment il a été formalisé.

Le théorème peut s'exprimer formellement ainsi[1] :

```
Lemma Smith_gcdr_spec m n (A : 'M_(m,n)) (d : seq R) k :
  \prod_(i < k) d`_i %= \big[gcdr/0]_f \big[gcdr/0]_g minor k f g A .
```

où le premier argument de la fonction `minor` est l'ordre du mineur. Les fonctions `f` et `g` jouent le rôle des ensembles d'indices I et J dans la notation $|A|_{IJ}$, plus précisément I correspond à l'image de `f` et J à l'image de `g`.

Soit donc `R` un anneau principal, une matrice `A : 'M[R]_(m,n)`, et une séquence `d` telle que `d` soit triée par la relation de division et que la matrice `diag_mxseq m n d` soit équivalente à `A`.

Le résultat se montre en deux étapes. Dans une première étape nous prouvons le théorème pour la matrice `diag_mx_seq m n d` (au lieu de `A`). Comme c'est une matrice diagonale, les seuls mineurs d'ordre k non nuls sont les mineurs principaux, c'est à dire les déterminants de sous-matrices dont la diagonale principale coïncide avec la diagonale de la matrice. Dans notre cas particulier, les mineurs principaux sont égaux au produit de k éléments de la séquence `d`. Comme chaque élément de la séquence divise le suivant, le pgcd des mineurs d'ordre k est donc le produit des k premiers éléments de la séquence.

Nous pouvons ensuite remplacer dans l'énoncé du théorème le membre de gauche `\prod_(i < k) d`_i` par l'expression suivante :

```
\big[gcdr/0]_f \big[gcdr/0]_g minor k f g (diag_mx_seq m n d)
```

Dans la seconde étape, il nous reste donc à prouver ceci :

```
\big[gcdr/0]_f \big[gcdr/0]_g minor k f g (diag_mx_seq m n d)
   %= \big[gcdr/0]_f \big[gcdr/0]_g minor k f g A
```

Cela se prouve par double divisibilité, comme les deux preuves se font de la même manière, nous montrons ici seulement le résultat suivant :

```
\big[gcdr/0]_f \big[gcdr/0]_g minor k f g A
   %| \big[gcdr/0]_f \big[gcdr/0]_g minor k f g (diag_mx_seq m n d)
```

D'après les propriétés du pgcd, il suffit de montrer que pour tout `f'` et `g'` nous avons :

[1] voir 1.2.5, et 1.2.7 pour les notations.

```
\big[gcdr/0]_f \big[gcdr/0]_g minor k f g A
  %| minor k f' g' (diag_mx_seq m n d)
```

Or nous savons que les matrices `A` et `diag_mx_seq m n d` sont équivalentes, il existe donc deux matrices `M` et `N` telles que `M *m A *m N = diag_mx_seq m n d`. En réécrivant cette égalité, il nous reste à prouver :

```
\big[gcdr/0]_f \big[gcdr/0]_g minor k f g A
  %| minor k f' g' (M *m A *m N)
```

Nous avons fait apparaître ici le mineur d'un produit de matrices, nous allons le transformer en une somme de produits de mineurs en utilisant la formule de Binet-Cauchy. Cette formule s'énonce comme suit :

$$\det(AB) = \sum_{\substack{I \in \mathcal{P}(\{1,\ldots,l\}) \\ \#|I|=k}} \det(A_I)\det(B_I)$$

où A est une matrice de taille $k \times l$ et B une matrice de taille $l \times k$. Ici $\det A_I = |A|_{\{1,\ldots,k\}I}$ et $\det B_I = |B|_{I\{1,\ldots,k\}}$. Ce résultat a été prouvé formellement par Vincent Siles [46]. On peut facilement ramener la formule de Binet-Cauchy à la formule suivante :

$$|AB|_{IJ} = \sum_K |A|_{IK}|B|_{KJ}$$

La formule ci-dessus s'écrirait formellement :

```
minor k f g (A *m B) = \sum_h (minor k f h A) * (minor k h g B)
```

Grâce à ce lemme nous pouvons remplacer `minor k f' g' (M *m A *m N)` par

```
\sum_h \sum_j (minor k f' h M) * (minor k h j A) * (minor k j g' N)
```

Or d'après les propriétés du pgcd, pour tout `h` et `j` nous avons :

```
\big[gcdr/0]_f \big[gcdr/0]_g minor k f g A %| minor k h j A
```

Donc le pgcd des mineurs d'ordre k de la matrice A divise tous les termes de la somme ci-dessus, ce que nous voulions démontrer.

Nous avons vu que les coefficients de la forme normale de Smith peuvent être choisis modulo la relation d'équivalence $\sim$. Cela implique en particulier, que, pour les matrices carrées, nous pouvons choisir un représentant de la forme normale de Smith d'une matrice A qui a le même déterminant. En effet, soit D la forme normale de Smith d'une matrice A. Nous savons que D est équivalent à la matrice A, il existe donc des matrices L et R inversibles telles que $D = LAR$. Le fait que L et R soient inversibles signifie que leur déterminant est également inversible. Donc si nous appelons D' la matrice D dans laquelle nous avons multiplié la première ligne par $(\det L * \det R)^{-1}$ alors :

- Les éléments diagonaux de D' sont triés pour la relation de division (car multiplier certains éléments par un inversible ne change pas les classes d'équivalences).

- La matrice D' est équivalente à la matrice A (car multiplier une ligne par un élément inversible est une opération réversible et donc conserve l'équivalence).

Formellement nous avons :

```
Definition Smith_seq m n (M : 'M[E]_(m,n)) :=
  let: (L,d,R) := (Smith _ M) in
  if d is a :: d' then (\det L)^-1 * (\det R)^-1 * a :: d' else nil.

Definition Smith_form m n (A : 'M[R]_(m,n)) :=
  diag_mx_seq m n (Smith_seq A).

Lemma det_Smith n (A : 'M[R]_n) : \det (Smith_form A) = \det A.
```

Le théorème d'unicité de la forme normale de Smith d'une matrice s'énonce formellement comme suit :

```
Lemma Smith_unicity m n (A : 'M[R]_(m,n)) (s : seq R) :
  sorted %| s -> equivalent A (diag_mx_seq m n s) ->
  forall i, i < minn m n -> s`_i %= (Smith_seq A)`_i.
```

2.3.3 Facteurs invariants

Dans la section 2.2.2, nous avons vu que nous pouvons déterminer si deux matrices carrées A et B à coefficients dans un corps sont semblables en comparant les formes normales de Smith de leur matrice caractéristique. Ensuite nous avons montré dans la section précédente un théorème qui permet de montrer l'unicité de la forme normale de Smith, mais dont nous nous servirons plus tard pour déterminer les éléments diagonaux de la forme normale de Smith d'une matrice par des calculs de pgcd. Nous allons présenter maintenant les facteurs invariants.

La forme normale de Smith de la matrice $XI - A$ est une matrice diagonale à coefficients polynômiaux. Les polynômes non-constants qui apparaissent sur la diagonale sont les facteurs invariants de la matrice A. Le théorème fondamental et le résultat d'unicité nous disent que les facteurs invariants sont des invariants de similitude, c'est-à-dire que deux matrices sont semblables si et seulement si elles ont les mêmes facteurs invariants.

Nous allons voir ici comment les facteurs invariants d'une matrice ont été définis formellement.

Rappelons que la forme normale de Smith d'une matrice à coefficients dans un anneau principal est unique modulo la relation d'équivalence $\sim$. Or dans notre contexte, l'anneau principal sur lequel nous travaillons est l'anneau des polynômes à coefficients dans un corps. Dans ce cas-ci, si p est un polynôme non nul alors il existe un unique représentant unitaire de la classe d'équivalence de p qui est p divisé par son coefficient dominant.

Nous pouvons donc définir une nouvelle séquence qui vérifie les mêmes spécifications que la séquence `Smith_seq` mais qui ne contient que des polynômes unitaires[1] :

```
Definition Frobenius_seq n (A : 'M_n) :=
  [seq (lead_coef p)^-1 *: p | p <- (Smith_seq (char_poly_mx A))].
```

Les facteurs invariants sont les polynômes non constants de cette séquence :

```
Definition invariant_factors n (A : 'M_n) :=
  [seq p : {poly R} <- (Frobenius_seq A) | 1 < size p].
```

2.4 Forme normale de Frobenius

La forme normale de Frobenius d'une matrice M est la matrice diagonale par blocs dont les blocs diagonaux sont les matrices compagnes des facteurs invariants de M.

Nous avons déjà vu comment sont définies formellement les matrices diagonales par blocs. Nous avons vu ensuite ce qu'était la forme normale de Smith, ce qui nous a permis de définir les facteurs invariants d'une matrice. Il nous reste donc à donner une définition formelle des matrices compagnes pour pouvoir définir la forme normale de Frobenius. Nous prouverons ensuite que toute matrice carrée à coefficients dans un corps est semblable à sa forme normale de Frobenius.

2.4.1 Matrices compagnes

La matrice compagne d'un polynôme unitaire $p = X^n + a_{n-1}X^{n-1} + \ldots + a_1X + a_0$ est la matrice suivante :

$$\mathcal{C}_p = \begin{pmatrix} 0 & \ldots & \ldots & 0 & -a_0 \\ 1 & \ddots & & \vdots & -a_1 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \ldots & 0 & 1 & -a_{n-1} \end{pmatrix}$$

Cette matrice est intéressante car p est à la fois son polynôme caractéristique et son polynôme minimal.

Formellement, si `p` est un polynôme, la taille de la matrice compagne de `p` est `(size p).-1`. Mais ici encore, si nous voulons utiliser des opérations génériques d'anneaux sur des matrices compagnes, il faut que leur taille soit de la forme `n.+1`. Pour cela nous définissons les matrices compagnes de sorte que leur type soit `(size p).-2.+1` :

[1] voir 1.1.5, 1.2.6 et 1.2.9 pour les notations.

```
Definition companion_mx (p : {poly R}) :=
  \matrix_(i,j < (size p).-2.+1)
     ((i == j.+1 :> nat)%:R + p`_i *+ ((size p).-2 == j)).
```

Nous avons `(size p).-2.+1 = (size p).-1` si et seulement si `1 < (size p)`, c'est-à-dire si `p` est un polynôme non constant. La définition de matrice compagne n'est donc valide que pour les polynômes non constants. En pratique ceci ne devrait pas être trop contraignant car les matrices compagnes de polynômes constants sont des matrices vides et sont donc inintéressantes dans un certain sens.

Malheureusement, avec cette définition de matrices compagnes, nous ne pouvons pas définir de matrice diagonale par blocs dont les blocs sont des matrices compagnes. En effet, le type de la fonction qui retourne les blocs diagonaux de la matrice est `forall n, nat -> 'M_n.+1` ; c'est donc une fonction qui prend un entier `n`, un entier et qui retourne une matrice de type `'M_n.+1`, mais si la matrice retournée est une matrice compagne d'un polynôme `p`, alors son type sera `'M_(size p).-2.+1` et non `'M_n.+1`.

Pour résoudre ce problème, nous définissons les matrices compagnes en deux étapes. Nous définissons d'abord une fonction `companion_mxn` qui prend en argument un entier `n` et un polynôme `p`, et qui retourne une matrice de type `'M_n` telle que si `n = (size p).-2.+1` alors la matrice retournée correspond à la matrice compagne du polynôme `p`. Nous pouvons donc ensuite définir la matrice compagne d'un polynôme `p` comme étant la matrice retournée par la fonction `companion_mxn` appliquée aux arguments `(size p).-2.+1` et `p` :

```
Definition companion_mxn n (p : {poly R}) :=
 \matrix_(i,j < n) ((i == j.+1 :>nat)%:R + p`_i *+ ((size p).-2 == j)).

Definition companion_mx (p : {poly R}) :=
  companion_mxn (size p).-2.+1 p.
```

Ainsi pour faire une matrice diagonale par blocs, nous pouvons utiliser la fonction `companion_mxn`, car `companion_mxn n.+1` a le type `'M_n.+1` qui est bien de la bonne forme. Les lemmes sur les matrices compagnes seront exprimés, eux, sur `companion_mx`.

Nous pouvons maintenant donner une définition formelle de la forme normale de Frobenius d'une matrice. Elle se présente de la manière suivante :

$$\begin{pmatrix} \mathcal{C}_{p_1} & & & 0 \\ & \mathcal{C}_{p_2} & & \\ & 0 & \ddots & \\ & & & \mathcal{C}_{p_k} \end{pmatrix}$$

où les polynômes p_i sont les facteurs invariants de la matrice. Formellement, nous la définissons comme suit :

```
Definition Frobenius_form n (A : 'M_n) :=
  let sizes := [seq (size p).-2 | p <- (invariant_factors A)] in
  let blocks n i := companion_mxn n.+1 (invariant_factors A)`_i in
   diag_block_mx sizes blocks.
```

2.4.2 De Smith à Frobenius

Maintenant que nous avons défini la forme normale de Frobenius d'une matrice, nous allons montrer ici que toute matrice sur un corps `F` est semblable à sa forme normale de Frobenius :

```
Lemma Frobenius n (A : 'M[F]_n.+1) : similar A (Frobenius_form A).
```

Comme nous l'avons annoncé, pour prouver ce résultat nous utilisons le théorème fondamental. Il nous faut donc montrer que pour toute matrice `A`, les matrices caractéristiques de `A` et `Frobenius_form A` sont équivalentes.

Soit une matrice A, nous allons partir de la matrice $XI - A$ et arriver à la matrice caractéristique de `Frobenius_form A` par transitivité de l'équivalence.

Nous avons vu que la séquence `Frobenius_seq A` respecte les spécifications de l'algorithme de Smith. Donc la matrice $XI - A$ est équivalente à la matrice `diag_mx_seq n.+1 n.+1 (Frobenius_seq A)`, qui se présente de la façon suivante :

$$\begin{pmatrix} 1 & & & & & \\ & \ddots & & & 0 & \\ & & 1 & & & \\ & & & p_1 & & \\ & 0 & & & \ddots & \\ & & & & & p_n \end{pmatrix}$$

où les p_i sont les facteurs invariants de la matrice A.

Ici nous allons utiliser le résultat suivant pour justifier le fait que nous pouvons permuter les éléments diagonaux :

```
Lemma similar_diag_mx_seq m n s1 s2 :
  m = n -> size s1 = m -> perm_eq s1 s2 ->
  similar (diag_mx_seq m m s1) (diag_mx_seq n n s2).
```

où `perm_eq s1 s2` exprime le fait que les séquence `s1` et `s2` ont les mêmes éléments mais permutés.

En effet, si deux matrices diagonales ont leurs éléments diagonaux permutés, alors la matrice de passage de l'une à l'autre est une matrice de permutation qui est inversible ; donc les deux matrices sont semblables, et donc à fortiori équivalentes. Nous pouvons ainsi permuter les éléments diagonaux de la matrice

précédente :

$$\begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & 0 & \\ & & 1 & & & & & & \\ & & & p_1 & & & & & \\ & & & & \ddots & & & & \\ & & & & & 1 & & & \\ & & & & & & \ddots & & \\ & 0 & & & & & & 1 & \\ & & & & & & & & p_n \end{pmatrix}$$

Si `pi` représente le polynôme p_i, alors `(size pi).-2` représente le nombre de 1 avant chaque p_i. Cette matrice peut être vue comme la matrice diagonale par blocs suivante :

$$\left(\begin{array}{cccc|c|cccc} 1 & & & & & & & & \\ & \ddots & & & & & 0 & & \\ & & 1 & & & & & & \\ & & & p_1 & & & & & \\ \hline & & & & \ddots & & & & \\ \hline & & & & & 1 & & & \\ & 0 & & & & & \ddots & & \\ & & & & & & & 1 & \\ & & & & & & & & p_n \end{array}\right)$$

Nous voulons maintenant montrer que cette matrice est équivalente à la matrice caractéristique de la forme normale de Frobenius de A. Nous utilisons d'abord le fait que la matrice caractéristique d'une matrice diagonale par blocs est la matrice diagonale par blocs des blocs caractéristiques :

```
Lemma char_diag_block_mx s (F : forall n, nat -> 'M[R]_n.+1) :
  s != [::] ->
  char_poly_mx (diag_block_mx s F) =
  diag_block_mx s (fun n i => char_poly_mx (F n i)).
```

Il ne nous reste donc plus qu'à montrer que la matrice précédente est équivalente à :

$$\begin{pmatrix} XI - \mathcal{C}_{p_1} & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & XI - \mathcal{C}_{p_n} \end{pmatrix}$$

Nous voulons donc montrer que deux matrices diagonales par blocs sont équivalentes. Pour cela, il nous suffit de prouver que les matrices sont équivalentes bloc

à bloc. C'est-à-dire que pour chaque indice i, la matrice $XI - \mathcal{C}_{p_i}$ est équivalente à :

$$\begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & p_i \end{pmatrix}$$

Pour montrer ce dernier résultat, nous utilisons le lemme `Smith_gcdr_spec` vu plus haut. En effet, pour tout `k` tel que `k < (size pi).-2`, on peut trouver une sous-matrice de la matrice $XI - \mathcal{C}_{p_i}$ ci-dessous n'ayant que des -1 sur la diagonale :

$$XI - \mathcal{C}_{p_i} = \begin{pmatrix} X & \dots & 0 & 0 & a_0 \\ -1 & \ddots & \vdots & \vdots & a_1 \\ 0 & \ddots & X & \vdots & \vdots \\ \vdots & \ddots & -1 & X & \vdots \\ 0 & \dots & 0 & -1 & X + a_{n-1} \end{pmatrix}$$

Donc si nous calculons la forme normale de Smith de la matrice ci-dessus comme nous l'avons fait dans l'exemple 1, nous remarquons que pour tout `k` strictement plus petit que `(size pi).-2`, il existe un mineur d'ordre `k` associé à 1 (car $(-1)^k \sim 1$), et donc le pgcd de ces mineurs est lui-même associé à 1. Ainsi tous les éléments diagonaux de la forme normale de Smith de la matrice $XI - \mathcal{C}_{p_i}$ sont égaux à 1, sauf le dernier qui lui est égal au déterminant de la matrice $XI - \mathcal{C}_{p_i}$, c'est-à-dire à p_i. Le fait qu'une matrice est équivalente à sa forme normale de Smith termine la démonstration.

2.5 Forme normale de Jordan

La forme normale de Jordan d'une matrice A est une matrice triangulaire supérieure dont les éléments diagonaux sont les racines du polynôme caractéristique de la matrice A (c'est-à-dire les valeurs propres de A). Pour que cette forme existe, il suffit que le polynôme caractéristique soit scindé à racines simples. Afin d'assurer cette condition, nous choisissons de travailler sur un corps algébriquement clos `F`.

Nous avons montré dans la section précédente qu'une matrice était semblable à sa forme normale de Frobenius. Nous allons ici dans un premier temps donner une définition formelle de la forme normale de Jordan. Nous montrerons ensuite que la forme normale de Frobenius d'une matrice est semblable à la forme normale de Jordan de celle-ci. Par transitivité de la similitude nous aurons donc qu'une matrice est semblable à sa forme normale de Jordan.

2.5.1 Définitions

On appelle bloc de Jordan la matrice de dimension n suivante :

$$J(\lambda, n) = \begin{pmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \dots & \dots & 0 & \lambda \end{pmatrix}$$

Formellement, cela se définit simplement de la manière suivante :

```
Definition Jordan_block lam n : 'M[F]_n :=
  \matrix_(i,j) (lam *+ (i == j) + (i.+1 == j)%:R).
```

La forme normale de Jordan est une matrice diagonale par blocs composée de blocs de Jordan[1] :

```
Definition Jordan_form n (A : 'M[R]_n.+1) :=
  let sp := root_seq_poly (invariant_factors A) in
  let sizes := [seq x.1 | x <- sp] in
  let blocks n i := Jordan_block (nth (0,0) sp i).2 n.+1 in
  diag_block_mx sizes blocks.
```

où la fonction `root_seq_poly` prend en argument une séquence de polynômes et retourne la concaténation des séquences des paires multiplicité/racine de chaque polynôme de la séquence. Par exemple si on a la séquence de polynômes suivante :

```
[:: ('X - (sqrt 2))^+ 4, ('X - (sqrt 2))^+ 4, ('X - 1)^+3 *('X - 2)]
```

alors la séquence retournée par `root_seq_poly` sera :

```
[:: (4,(sqrt 2))] ++ [:: (4,(sqrt 2))] ++ [:: (3,1), (1,2)] =
[:: (4,sqrt2), (4,sqrt2), (3,1), (1,2)]
```

Dans la suite nous expliquons le passage de la forme normale de Frobenius à celle de Jordan. Cela permettra de mieux comprendre la définition donnée ci-dessus.

2.5.2 De Frobenius à Jordan

Soit A une matrice à coefficients dans un corps clos `F`. Nous avons déjà prouvé que la matrice A est semblable à sa forme normale de Frobenius. Nous allons maintenant montrer que la forme normale de Frobenius de A est semblable à sa forme normale de Jordan. Ce qui nous permettra de montrer le lemme suivant :

```
Lemma Jordan n (A : 'M[F]_n.+1) : similar A (Jordan_form A).
```

La preuve de ce lemme utilise deux résultats intermédiaires. Le premier résultat intermédiaire est le lemme suivant :

[1] voir 1.1.1 et 1.1.5 pour les notations.

Lemme 2. *Soit q un polynôme, soit $(q_i)_{1\leq i\leq m}$ une famille de polynômes unitaires premiers entre eux deux à deux telle que $q = q_1 \dots q_m$, alors la matrice $\mathcal{C}_q$ est semblable à la matrice :*

$$\begin{pmatrix} \mathcal{C}_{q_1} & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & \mathcal{C}_{q_m} \end{pmatrix}$$

Montrons d'abord ce lemme pour $m = 2$. Soit donc deux polynômes unitaires q_1 et q_2 premiers entre eux, et soit $q = q_1 q_2$. Nous voulons montrer que la matrice $\mathcal{C}_q$ est semblable à la matrice :

$$\begin{pmatrix} \mathcal{C}_{q_1} & 0 \\ 0 & \mathcal{C}_{q_2} \end{pmatrix}$$

Là encore nous utilisons le théorème fondamental, pour se ramener à montrer l'équivalence entre $XI - \mathcal{C}_q$ et la matrice :

$$\begin{pmatrix} XI - \mathcal{C}_{q_1} & 0 \\ 0 & XI - \mathcal{C}_{q_2} \end{pmatrix}$$

D'après ce que nous avons déjà vu cela revient à montrer que la matrice

$$\begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & q \end{pmatrix}$$

est équivalente à la matrice

$$\begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & 0 & \\ & & 1 & & & & & \\ & & & q_1 & & & & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & 0 & & & & & 1 & \\ & & & & & & & q_2 \end{pmatrix}$$

Autrement dit, il faut montrer que le seul facteur invariant de la matrice ci-dessus est le polynôme q. Pour cela nous allons utiliser une fois de plus le lemme `Smith_gcdr_spec`. En effet, pour tous les ordres k strictement plus petit que la taille de la matrice ci-dessus, on peut trouver un mineur principal égal à q_1, et un autre égal à q_2, et comme q_1 et q_2 sont premiers entre eux, le pgcd des mineurs d'ordre k est égal à 1. Donc les éléments diagonaux de la forme normale de Smith

de la matrice ci-dessus sont donc tous égaux à 1, sauf le dernier qui correspond au déterminant de la matrice ci-dessus qui est $q_1 q_2 = q$. Le lemme général en découle directement par récurrence.

Le deuxième résultat intermédiaire est celui-ci :

Lemme 3. *Soit $\mathbb{K}$ un corps. Soit $\lambda \in \mathbb{K}$ et $n \in \mathbb{N}$. alors la matrice $\mathcal{C}_{(X-\lambda)^n}$ est semblable à la matrice $J(\lambda, n)$.*

Ici nous pouvons directement donner la matrice de passage P définie comme suit :

$$P_{ij} = \binom{j-1}{n-i} \lambda^{(i+j)-(n+1)}$$

Mais la vérification formelle est assez longue. Sinon nous pouvons utiliser une nouvelle fois le théorème fondamental, et montrer que $XI - \mathcal{C}_{(X-\lambda)^n}$ est équivalente à la matrice $XI - J(\lambda, n)$. D'après ce que nous savons déjà sur les matrices compagnes, nous voulons montrer que la matrice :

$$\begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & (X-\lambda)^n \end{pmatrix}$$

est équivalente à la matrice :

$$\begin{pmatrix} X-\lambda & -1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -1 \\ 0 & \dots & \dots & 0 & X-\lambda \end{pmatrix}$$

Ici encore, nous allons utiliser le lemme `Smith_gcdr_spec`. En effet, pour tout $k < n$, nous pouvons trouver dans la matrice ci-dessus un mineur d'ordre k égal à $(-1)^k$, et donc associé à 1. Donc par les mêmes raisonnement que précédemment, le seul facteur invariant de la matrice ci-dessus est $(X-\lambda)^n$.

Maintenant nous pouvons passer à la preuve de la similitude entre les formes normales de Frobenius et de Jordan d'une matrice.

Comme nous sommes sur un corps clos, alors chaque facteur invariant p_i de la matrice A peut se décomposer comme suit :

$$p_i = \prod_{j=1}^{m_i} (X - \lambda_{ij})^{\mu_{ij}}$$

où les λ_{ij} sont les racines de p_i et les μ_{ij} leur multiplicité.

Donc d'après le lemme 2 montré ci-dessus, chaque bloc $\mathcal{C}_{p_i}$ de la forme normale de Frobenius est semblable à la matrice :

$$\begin{pmatrix} \mathcal{C}_{(X-\lambda_{i1})^{\mu_{i1}}} & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & \mathcal{C}_{(X-\lambda_{im_i})^{\mu_{im_i}}} \end{pmatrix}$$

et d'après le lemme 3, chacun des blocs $\mathcal{C}_{(X-\lambda_{ij})^{\mu_{ij}}}$ de la matrice ci-dessus est semblable au bloc de Jordan $J(\lambda_{ij}, \mu_{ij})$.

Les paires (λ_{ij}, μ_{ij}) sont les paires qui se trouvent dans la séquence retournée par la fonction `root_seq_poly` appliquée à la séquence des facteurs invariants. Cela explique la définition de la forme normale de Jordan donnée plus haut, et démontre aussi que la forme normale de Frobenius de A est semblable à la forme normale de Jordan de A.

2.5.3 Diagonalisation

En plus de la forme normale de Jordan, la théorie des facteurs invariants permet de donner également un théorème de diagonalisation. Une matrice est diagonalisable, c'est-à-dire semblable à une matrice diagonale, si son polynôme minimal est scindé et à racines simples. Nous allons voir ici comment nous prouvons ce résultat.

Commençons par montrer le théorème suivant :

Théorème 4. *Soit A une matrice à coefficients dans un corps. Si les polynômes $p_1 \dots p_k$ sont les facteurs invariants de la matrice A tels que $p_1 \mid p_2 \mid \dots \mid p_k$, alors le polynôme p_k est le polynôme minimal de la matrice A.*

Soit donc A une matrice à coefficients dans un corps. Comme la matrice A et la forme normale de Frobenius de A sont semblables, ces deux matrices ont le même polynôme minimal. Donc montrer que p_k est le polynôme minimal de A revient à montrer que p_k est le polynôme minimal de la forme normale de Frobenius de A.

Pour cela, nous allons montrer d'une part que p_k est un polynôme annulateur de la forme normale de Frobenius de A, et d'autre part que p_k divise n'importe quel polynôme annulateur de la forme normale de Frobenius de A.

Un polynôme p est annulateur de la matrice A si $p(A) = 0$. Cela peut s'exprimer formellement grâce à la fonction `horner_mx` de la bibliothèque SSReflect comme suit :

```
horner_mx A p = 0
```

Dans cette preuve, nous utiliserons le fait qu'appliquer un polynôme à une matrice diagonale par blocs revient à appliquer le polynôme à chacun des blocs. Ce qui s'exprime formellement de la façon suivante.

```
Lemma horner_mx_diag_block (p : {poly R}) s F :
  s != [::] ->
  horner_mx (diag_block_mx s F) p =
  diag_block_mx s (fun n i => horner_mx (F n i) p).
```

Donc pour montrer que le polynôme p_k annule la forme normale de Frobenius de A, il faut et il suffit que p_k annule chaque matrice compagne des facteurs invariants.

Soit donc p_i un facteur invarant de la matrice A. D'après les propriétés des matrices compagnes, nous savons que p_i est le polynôme minimal de la matrice $\mathcal{C}_{p_i}$. De plus comme les facteurs invariants se divisent successivement, nous savons également que $p_i \mid p_k$. Donc p_k est un polynôme annulateur de $\mathcal{C}_{p_i}$.

Maintenant il reste à montrer que p_k divise tous les polynômes annulateurs de la forme normale de Frobenius de A.

Soit donc Q un polynôme annulateur de la forme normale de Frobenius de A. Alors nous savons que Q est un polynôme annulateur de chacun des blocs de la forme normale de Frobenius de A. En particulier Q annule $\mathcal{C}_{p_k}$, or nous savons également que p_k est le polynôme minimal de la matrice $\mathcal{C}_{p_k}$. Donc $p_k \mid Q$.

Le polynôme p_k étant un polynôme annulateur de la forme normale de Frobenius de A qui divise tous les autres polynômes annulateurs est donc le polynôme minimal de la forme de Frobenius de A et donc de A.

Ainsi nous avons d'une part que le polynôme p_k est le polynôme minimal de la matrice A, et d'autre part que tous les facteurs invariants divisent p_k. Autrement dit tous les facteurs invariants de la matrice A divisent son polynôme minimal. Donc si le polynôme minimal de A est scindé à racines simples, alors c'est aussi le cas des facteurs invariants. Or la taille des blocs de la forme normale de Jordan de A sont les multiplicités des racines des facteurs invariants. Donc dans le cas où le polynôme minimal de A est scindé à racines simples, tous les blocs de la forme normale de Jordan de la matrice A sont de taille 1, c'est donc une matrice diagonale. Ce qui nous donne le théorème suivant :

```
Lemma ex_diagonalization n (A : 'M[R]_n.+1) :
   uniq (root_seq (mxminpoly A)) ->
  {s | similar A (diag_mx_seq n.+1 n.+1 s)}.
```

où `root_seq` est la fonction qui retourne la séquence des racines d'un polynôme et `uniq` est un prédicat booléen qui prend la valeur `true` si la séquence passée en argument n'a pas de doublons.

2.6 Conclusion

Nous avons utilisé ici les matrices de la bibliothèque SSReflect. Le type de ces matrices dépend de leurs dimensions, et cela a posé quelques problèmes. Cependant nous avons vu qu'il y avait des outils (comme `conform_mx`) qui ont permis de résoudre certains de ces problèmes.

Au départ, pour prouver certains lemmes de similitude, nous donnions directement les matrices de passages, et nous avions de longues preuves faites de traitements par cas exhaustifs. L'idée d'utiliser de manière assez systématique le théorème fondamental et le lemme `Smith_gcdr_spec` à grandement simplifié les preuves.

Ce travail, en plus de la formalisation de la forme normale de Jordan d'une matrice, a apporté une bibliothèque sur les matrices diagonales par blocs, sur les matrices compagnes, et sur les propriétés de similitude et d'équivalence de matrices. Ce sont des concepts importants de l'algèbre linéaire utilisés dans d'autres contextes mathématiques que celui de ce chapitre.

Nous avons aussi montré la formalisation de la forme normale de Frobenius d'une matrice. L'avantage de cette forme normale est que ses coefficients sont dans le même corps que les coefficients de la matrice de départ. Contrairement à la forme de Jordan, où l'on peut avoir des coefficients dans la clôture algébrique.

2.7 Travaux reliés

Le développement de la forme normale de Smith que nous avons utilisé dans ce chapitre fait parti d'un projet qui s'appelle CoqEAL(Coq Efficient Algebra Library)[1]. Le but de ce projet est de faire des algorithmes effectifs d'algèbre linéaire en utilisant l'extension SSReflect de Coq.

Dans les autres assistants de preuve, il existe d'autres travaux qui ont été faits sur les matrices. En HOL, les matrices sont vues comme des vecteurs de vecteurs [23]. En ACL2 elles sont représentées soit par des listes de vecteurs de même dimensions [49] soit par des tableaux [17]. En Isabelle, ce sont des fonctions telles que l'ensemble des images non nulles de ces fonctions soit fini [36].

À ma connaissance, seul l'assistant de preuve Mizar possède des travaux sur les matrices diagonales par blocs et sur la forme de Jordan d'une matrice [41]. Mizar est un assistant de preuve basé sur la théorie des ensembles contrairement à Coq qui est basé sur la théorie des types.

En Mizar, une séquence finie est une fonction dont le domaine de définition est l'ensemble des entiers de 1 à n pour un certain $n \in \mathbb{N}$. Une matrice M à m lignes et n colonnes est une séquence finie de séquences finies de longueur m tel que toute séquence finie de l'image de M soit de longueur n [27].

Une matrice diagonale par blocs est construite à partir d'une séquence finie de matrices carrées. Elle est définie comme étant la matrice dont l'élément (i, j) est 0[2] si l'on se trouve en dehors des blocs et l'éléments $f(i), g(j)$ d'une matrice de la séquence sinon (pour certaines fonctions f et g) [39].

[1] http://www.maximedenes.fr/content/coqeal-coq-effective-algebra-library

[2] En fait, la définition de Mizar de matrice diagonale par block permet de remplir la matrice hors des blocs diagonaux par un élément d que l'on peut passer en paramètre.

Chapitre 3

Topologie

Dans ce chapitre, nous allons présenter un résultat important de topologie générale qui est le théorème de Bolzano-Weierstraß. Ce théorème porte sur les valeurs d'adhérence des suites dont les valeurs sont dans un ensemble compact.

Tous le travail sur la topologie et sur la théorie des ensembles est formalisé dans un cadre fortement classique, c'est-à-dire avec l'axiome du tiers exclu et l'axiome du choix. Nous avons besoin de ces axiomes pour prouver le théorème de Bolzano-Weierstraß.

Il existe dans la bibliothèque standard de Coq un fichier `Rtopology.v`[1] qui contient déjà certaines notions que nous allons présenter dans ce chapitre. Mais ces notions ne sont définies que pour les réels de Coq. Nous présentons ici un travail plus général.

Nous commencerons par voir quelques points de la théorie des ensembles dont nous aurons besoin pour la formalisation de la topologie générale. Nous verrons ensuite comment sont définies les structures d'espaces métriques et topologiques, et nous terminerons par une présentation du théorème.

3.1 Théorie des ensembles

Pour parler d'ensembles ouverts, fermés ou compacts, il faut d'abord définir ce qu'est un ensemble. La notion d'ensemble utilisée dans cette formalisation est celle que l'on peut trouver dans la bibliothèque `Sets` de la bibliothèque standard de Coq[2], à savoir qu'un ensemble d'éléments de type `T` est identifié à un prédicat de type `T -> Prop`. Cependant nous n'utiliserons pas la théorie des ensembles ainsi développée dans Coq, car par exemple les définitions d'union, d'intersection, ou de complémentaire sur les ensembles sont intoduites explicitement à l'aide de types inductifs, alors que ces notions peuvent être définies comme une réinterprétation des opérateurs "et", "ou", "non" de la logique propositionnelle.

L'égalité utilisée entre les ensembles est l'égalité par inclusion mutuelle, mais

[1] http://coq.inria.fr/distrib/current/stdlib/Coq.Reals.Rtopology.html
[2] http://coq.inria.fr/distrib/current/stdlib/

nous avons tout de même besoin de la propriété d'extensionnalité. Plutôt que d'ajouter cette propriété comme axiome, nous allons voir comment faire en sorte d'utiliser l'égalité ensembliste comme l'égalité de Leibniz. Nous nous attarderons ensuite sur la définition de famille d'ensembles car nous en aurons besoin pour définir ce qu'est un ensemble compact.

3.1.1 Réécriture

La tactique `rewrite` en Coq permet à partir d'une hypothèse de type `x = y`

(où `=` représente l'égalité de Leibniz) de changer des occurences de `x` par des occurences de `y` et vice versa.

Si on remplace l'égalité de Leibniz par une relation d'équivalence, on peut aussi utiliser la tactique `rewrite` si `x` et `y` apparaissent dans une fonction dont on a montré qu'elle était compatible avec la relation d'équivalence, c'est-à-dire que sa valeur ne dépend pas du représentant choisi dans la classe d'équivalence. C'est ce que nous allons voir ici avec comme relation d'équivalence l'égalité entre ensembles.

L'égalité entre ensembles est définie comme suit :

```
Variable T : Type.

Definition eqset (A B : T -> Prop) :=
  forall x, x 'in A <-> x 'in B.
Notation "A =s B" := (eqset A B).
```

où `x 'in A` est une notation pour `A x`.

C'est une relation d'équivalence. Pour que le système Coq puisse la reconnaître comme telle, il faut la déclarer de la manière suivante :

```
Add Relation _ eqset
reflexivity proved by eqset_refl
symmetry proved by eqset_sym
transitivity proved by eqset_trans
as eqset_rel.
```

où `eqset_refl`,`eqset_sym`, et `eqset_trans` sont respectivement les lemmes de reflexivité, symétrie et de transitivité de la relation `eqset`.

Puis il faut indiquer à Coq quelles sont les fonctions sous lesquelles on peut réécrire avec la relation `eqset`, c'est-à-dire les fonctions qui sont compatibles avec la relation `eqset`. Pour cela on utilise les mots clefs `Add Morphism`. Prenons comme exemple l'inclusion. L'inclusion est définie comme suit :

```
Definition include (A B : T -> Prop) :=
  forall x, x 'in A -> x 'in B.
Notation "A <s B" := (include A B).
```

Nous déclarons au système que nous voulons pouvoir réécrire sous l'inclusion comme suit :

```
Add Morphism include with signature eqset ==> eqset ==> iff
  as include_mor.
Proof.
...
Qed.
```

Une manière de lire la signature est que si deux ensembles vérifient la relation `eqset`, et que deux autres ensembles vérifient également la relation `eqset`, alors nous avons deux propositions d'inclusions qui vérifient la relation `iff`, autrement dit qui sont équivalentes. Le système nous demande donc de prouver l'énoncé suivant :

```
forall x y : T -> Prop,
x =s y -> forall x0 y0 : T -> Prop, x0 =s y0 -> (x <s x0 <-> y <s y0)
```

Si par exemple au lieu de `iff`, nous avions mis `eq`, alors nous aurions dû prouver une égalité au lieu d'une équivalence.

Une fois que l'on a donné un terme de preuve de cet énoncé, le système crée deux objets `include_mor` et `include_mor_Proper` qui ont tous les deux le type du terme de preuve vu ci-dessus. Ceci vient du fait que le système de réécriture avec une relation d'équivalence autre que l'égalité de Leibniz utilise le système des classes de types fait par Matthieu Sozeau et Nicolas Oury [47]. En dehors de la section de définition, ces propriétés d'équivalence et de morphismes peuvent être activées par les commandes :

```
Existing Instance eqset_rel.
Existing Instance include_mor_Proper.
```

Toutes les définitions usuelles sur les ensembles sont compatibles avec la relation d'équivalence `eqset`. Ainsi nous pouvons faire des réécritures avec l'égalité ensembliste comme avec l'égalité de Leibniz (à part peut-être dans de rares cas). Nous n'avons donc pas besoin de l'axiome d'extensionnalité.

3.1.2 Les familles d'ensembles

Une famille d'ensembles $(F_i)_{i \in I}$ est caractérisée par un ensemble d'indices I, et les ensembles F_i indexés par I. Cela est formalisé en Coq à l'aide d'une structure :

```
Variable T : Type.
Variable Ti : eqType.

Structure family : Type := mkfamily {
    ind : Ti -> Prop;
    F :> Ti -> T -> Prop
  }.
```

Dans cette définition, il n'y a pas de dépendance entre la fonction `F` et l'ensemble d'indices `ind`. Donc dans certaines définitions et certains lemmes, il faudra préciser que l'on ne s'intéresse qu'aux indices appartenant à l'ensemble d'indices

de la famille. En effet, si nous avons un indice `i : Ti` qui n'appartient pas à l'ensemble `ind`, alors `F i` est un ensemble qui n'appartient pas à la famille, et donc on ne voudrait pas que cet ensemble soit pris en compte dans certaines preuves ou définitions.

Les autres définitions dont nous aurons besoin pour la définition d'ensemble compact sont celles de sous-famille, de famille finie et de recouvrement d'un ensemble par une famille.

Une sous-famille est définie comme un prédicat qui prend deux familles, et qui vérifie que l'ensemble d'indices de l'une est bien un sous-ensemble de l'ensemble d'indices de l'autre, et que sur ce sous-ensemble d'indices commun, les ensembles des deux familles sont les mêmes :

```
Definition sub_family f g :=
  ind g <s ind f /\ forall i, i 'in (ind g) -> f i =s g i.
```

Pour les familles finies, nous avons le choix entre définir un prédicat comme pour les sous-familles, ou définir un type "famille finie". Dans le premier cas, à chaque usage d'une définition sur les familles finies, nous devrons fournir une preuve que la famille passée en argument est finie. Alors que dans le second cas, l'information qu'une famille est finie est donnée par le typage. Cette dernière option permet d'avoir des définitions qui ne dépendent pas d'un terme de preuve. Une famille est finie si son ensemble d'indices est un ensemble fini :

```
Definition finite_set (T1 : eqType) (A : T1 -> Prop) :=
  exists s : seq T1, A =s (fun x => x \in s).

Structure finite_family := Ff {
   fam :> family;
   _ : finite_set (ind fam)
}.
```

Une fois que nous avons défini les familles, nous pouvons appliquer des opérations dessus, comme l'union ou l'intersection des éléments d'une famille. Nous donnons ici uniquement l'opération qui nous intéresse pour la définition de recouvrement :

```
Definition union_fam (f : family) (x : T) :=
  exists i, i 'in (ind f) /\ x 'in (f i).

Definition cover (A : T -> Prop) (f : family) := A <s union_fam f.
```

Nous avons maintenant tous les éléments pour nous intéresser à la topologie. Dans un premier temps nous allons voir comment sont définies les structures topologiques, enfin nous terminerons par la formalisation du théorème de Bolzano-Weierstraß.

3.2 Structures Topologiques

Les espaces topologiques, métriques, normés ou munis d'un produit scalaire, forment une certaine hiérarchie dans le sens où avec un produit scalaire on peut construire une norme, avec une norme on peut construire une distance, et avec une distance on peut construire une topologie, et tout cela de manière canonique. Cette hiérachie n'est pas héréditaire comme la hiérachie des structures algébriques. Dans une hiérarchie héréditaire, chaque structure de la hiérarchie est basée sur la structure précédente, à laquelle on a ajouté certaines lois ou propriétés (par exemple la structure d'anneau est basée sur la structure de groupe à laquelle on a ajouté une loi de multiplication qui vérifie certaines propriétés). Alors qu'ici toutes les structures d'espaces topologique, métrique, etc, sont indépendantes, mais chacune est une instance des structures précédentes dans la hiérarchie. Pour implémenter ces structures, nous utiliserons le modèle que l'on trouve principalement dans le fichier `ssralg.v` de la bibliothèque SSReflect. Nous allons voir ici pourquoi ce modèle est préférable plutôt qu'une approche naïve que nous présenterons d'abord.

3.2.1 Définition

Nous allons d'abord donner les définitions d'espace topologique et d'espace métrique dans des structures naïves qui contiennent simplement les axiomes qui définissent ces espaces.

Espace Topologique

Si $\mathcal{T}$ est un ensemble de points, alors une topologie sur $\mathcal{T}$ est la donnée d'un ensemble de parties appelées “ensembles ouverts” qui doit être stable par union, stable par intersection finie et qui doit contenir l'ensemble vide et $\mathcal{T}$. Un espace topologique est un ensemble de points munis d'une topologie.

L'ensemble des ouverts est un ensemble d'ensembles, et est donc représenté par un élément de type `(T -> Prop) -> Prop`. L'union de tous les ensembles d'un ensemble d'ensembles est définie comme suit :

```
Definition union_s (E : (T -> Prop) -> Prop)
  (f : (T-> Prop) -> (T -> Prop)) (x : T) :=
  exists A, A 'in E /\ x 'in (f A).
```

Par exemple, si E est un ensemble d'ensembles et f une fonction sur les ensembles, alors `union_s E f` représente l'ensemble suivant :

$$\bigcup_{A \in E} f(A)$$

Donc pour faire l'union des ensembles de E nous utiliserons la fonction identité pour f.

On peut maintenant définir la structure d'espace topologique simplement comme suit :

```
Structure Topologicalspace : Type := mkTS {
    T_Space : Type;
    open : (T_Space -> Prop) -> Prop;
    _ : open {ø : T_Space};
    _ : open {full : T_Space};
    _ : forall (E : (T_Space -> Prop) -> Prop), E <s open ->
          open (union_s E id);
    _ : forall A B, open A -> open B -> open (inter A B);
    _ : forall A B, A =s B -> (open A <-> open B)
  }.
```

Le dernier champ de la structure est la propriété d'extensionnalité pour les ouverts. Les ouverts étant définis axiomatiquement, cette propriété utile ne peut être démontrée, et a donc dû être ajoutée à la définition d'espace topologique.

Espace Métrique

Un espace métrique est un ensemble de points muni d'une distance. Il existe déjà une définition de cette structure dans le fichier `Rlimit.v` de la bibliothèque standard de Coq[1]. Dans cette définition le type de retour de la fonction distance est le type des réels de Coq. Ici on s'autorise à ce que ce soit une structure ordonnée plus générale :

```
Structure Metricspace (Rn : numDomainType) := mkMS {
   T_metric : Type.
   dist : T_metric -> T_metric -> Rn;
    _ : forall x y, 0 <= dist x y ;
    _ : forall x y, dist x y = dist y x;
    _ : forall x y, reflect (x = y) (dist x y == 0);
    _ : forall x y z, dist x y <= dist x z + dist z y
   }.
```

L'espace métrique prend en paramètre le type de retour de la distance `dist`. Ce type de retour doit être une structure ordonnée. Au début ce paramètre n'existait pas. Le type de retour de la fonction distance était directement le type des réels de Coq. Le problème était que l'on ne pouvait pas déclarer la structure `numDomainType` par exemple comme une instance de la structure d'espace métrique. Car la fontion de distance que l'on définit pour cela utilise la fonction `'|.|` dont le type de retour est une instance abstraite de la structure `numDomainType` et non le type concret des réels de Coq. Mettre le type de retour de la fonction distance en paramètre de la structure d'espace métrique permet d'implémenter la structure d'espace métrique sur des types plus abstraits comme le type des corps réels clos, des corps archimédiens, ou des complexes.

Pour un ensemble `E` de type `Metricspace`, on utilisera la notation suivante :

```
Notation d := (dist E).
```

[1] http://coq.inria.fr/distrib/current/stdlib/Coq.Reals.Rlimit.html

La principale notion utilisée pour étudier les propriétés métriques et topologiques d'un espace métrique est la notion de boule. Une boule est un ensemble défini à partir d'un élément de l'espace métrique qui sera appelé centre, et d'un rayon représenté par un nombre strictement positif. Pour éviter d'avoir dans la définition de boule une preuve que le rayon est strictement positif, on peut définir un type des nombres strictement positifs comme suit :

```
Structure posDomain (T: numDomainType) :=
   PosD { sort :> T ; _ : 0 < sort }.
Notation "{ 'posD' T }" := (posDomain T).
```

Ceci permet de définir les boules comme suit :

```
Variable Rn : numDomainType.
Variable E : (MetricSpace Rn).

Definition open_ball (x: E) (r : {posD Rn}) (y : E) := d x y < r.
Notation "''B' ( x , r )" := (open_ball x r).
```

Nous allons voir maintenant comment inférer les instances de structures canoniques avec les définitions ci-dessus. Les structures que nous définirons ensuite contiendrons les mêmes axiomes. Ce qui changera sera seulement la façon d'implémenter les structures.

3.2.2 Approche naïve

Pour déclarer dans le système Coq qu'un espace métrique possède canoniquement une structure d'espace topologique il faut d'abord définir ce qu'est un ensemble ouvert dans un espace métrique :

```
Variable Rn : numDomainType.
Variable E : (Metricspace Rn).

Definition open_met (A : E -> Prop) :=
  forall x, x 'in A -> exists r, 'B(x,r) <s A.
```

Ensuite, après avoir montré que cette définition d'ouverts vérifie toutes les propriétés qui apparaissent dans la structure d'espace topologique, nous pouvons construire à l'aide du constructeur `mkTS` de la structure `TopologicalSpace` l'espace topologique correspondant. Nous déclarons ensuite que cette construction est canonique avec le mot clé `Canonical` :

```
Definition EspMet_Top := mkTS empty_in_open_met full_in_open_met
  union_open_met inter_open_met ext_open_met.

Canonical EspMet_Top.
```

On pourrait procéder de la même manière pour montrer qu'un espace normé possède une structure d'espace métrique, et qu'un espace avec un produit scalaire possède une structure d'espace normé.

Nous nous trouvons maintenant confrontés à deux limitations du système de structures canoniques de COQ. La première est que les déclarations de structures canoniques ne sont pas transitives. La deuxième est que pour enregistrer les projections canoniques, seul l'élément de tête est pris en compte (i.e. si un type s'exprime comme `f (...)` pour une certaine fonction `f`, seule la fonction de tête `f` sera enregistrée dans la table de projection). Cela pose certains problèmes car l'idée de canonicité sous-entend l'idée d'unicité, et donc on ne peut pas déclarer deux structures canoniques différentes sur le même type, or si nous avons deux types différents, mais qui sont le résultat de l'appel d'une même fonction mais pour des arguments différents, le système de structures canoniques ne fait pas la différence.

Nous allons voir plus concrètement quels types de problèmes nous avons rencontrés, et certains moyens de résoudre ces problèmes. Prenons un exemple simple avec trois "étages" : une structure topologique, une structure d'espace métrique dont on a montré qu'elle était une instance de la structure topologique, et un type concret qui est le type des nombres réels de COQ (que l'on notera `R` par la suite) dont on veut montrer que c'est un espace métrique, et donc par la même occasion un espace topologique.

Pour montrer que l'on a une instance d'une structure, on procède toujours de la même manière. Il faut utiliser le constructeur de la structure, et lui donner en argument tous les champs de la structure. Ici `mkMS` est le constructeur de la structure d'espace métrique, les autres arguments sont les preuves que la distance sur `R` vérifie les axiomes de la définition de distance (ces lemmes sont déjà principalement prouvés dans la bibliothèque standard de COQ) :

```
Definition R_Met := mkMS R_dist_pos R_dist_sym R_dist_refl R_dist_tri.
Canonical R_Met.
```

Nous avons donc montré d'une part que les espaces métriques ont une topologie canonique, d'autre part que `R` avec la distance usuelle avait une structure d'espace métrique. Ce que nous aimerions maintenant, c'est utiliser ces deux résultats pour déclarer au système que `R` possède une structure d'espace topologique sans devoir démontrer toutes les propriétés de la structure d'espace topologique.

Une première idée serait d'utiliser la fonction `EspMet_Top` que l'on a définie plus haut, car c'est justement une fonction qui prend en argument un espace métrique et qui retourne l'espace topologique associé :

```
Definition R_Top := Eval hnf in EspMet_Top R_Met.
```

Le problème est qu'en réalité `R_Top` n'est pas un espace topologique sur `R`, mais sur l'espace métrique `R_Met`. Or il existe déjà une structure canonique des espaces métriques vers les espaces topologiques qui est `EspMet_Top`, donc si on déclare `R_Top` comme étant une instance canonique, le système va renvoyer un message d'avertissement disant que la projection canonique n'a pas été prise en compte car elle est redondante avec la déclaration de `EspMet_Top`.

Si nous faisons afficher au système la fonction `R_Top` avec tous ses arguments nous aurons :

```
R_Top = @mkTS (T_metric realnumDomainType R_Met) (@open_met ..) ...
```

où `realnumDomainType` est la structure de `numDomainType` sur les réels de Coq. Pour des raisons de clarté, `rn` désignera dans la suite `realnumDomainType`.

Donc si nous essayons de déclarer `R_Top` comme une instance canonique, le système va essayer d'enregistrer la projection de `T_Space` vers l'opérateur de tête de `T_metric rn R_Met` qui est `T_metric`. Or il existe déjà une projection canonique de `T_Space` vers `T_metric`, c'est celle qui correspond à `EspMet_Top`. Pourtant `T_metric rn R_Met` est réductible à `R`. Pour forcer le système Coq à faire cette réduction, on peut utiliser la tactique `unfold` avec la commande `Eval`. La tactique `unfold` va permettre de déplier certaines définitions, en particulier la définition de l'opérateur de tête, ce qui va permettre d'enregistrer la projection canonique. Dans notre exemple, on l'utilisera comme suit :

```
Definition R_Top :=
  Eval unfold EspMet_Top, T_metric, R_Met in EspMet_Top R_Met.
```

Ici, on demande à la tactique `unfold` de déplier les définitions de `EspMet_Top`, `T_metric`, et de `R_Met`. Si on affiche de nouveau `R_Top` nous pouvons remarquer que `T_metric rn R_Met` a été réduit à `R`. Cette réduction fait que `R_Top` est devenu une structure d'espace topologique directement sur `R` (et pas sur `T_metric rn R_Met`). Il n'y a donc plus aucun problème pour déclarer à Coq que l'on veut que cette instance soit canonique :

```
Canonical R_Top.
```

Pour l'instant seules les structures d'espace métrique et d'espace topologique sont implémentées. Ici nous avons donc un exemple simple, avec seulement trois niveaux, mais si l'on veut rajouter par exemple les structures d'espace normé ou avec un produit scalaire, le nombre de définitions à déplier va être plus grand, et l'on n'a pas forcément envie de dire manuellement au système quelles définitions il faut déplier pour la réduction d'un terme. Nous allons donc voir maintenant une façon différente d'implémenter les structures.

3.2.3 Approche SSReflect

Le modèle de structure que nous allons voir ici est le même que celui utilisé principalement dans la bibliothèque SSReflect. Ici nous expliquerons brièvement ce modèle, pour une explication plus complète, le lecteur pourra se référer à la thèse de François Garillot [14]. Le problème que l'on a rencontré ci-dessus est dû au fait que le type de base (par exemple les champs `T_space` ou `T_metric` dans les structures ci-dessus) est interne à la structure (i.e. c'est un des champs de la structure), et donc il est automatiquement déterminé par le système à partir des autres champs. Dans l'exemple ci-dessus, la fonction `EspMet_Top` construit directement la structure d'espace topologique, et nous n'avons pas de moyen d'influer directement sur la valeur des champs de la structure. Par exemple on aurait bien aimé pouvoir dire que l'on veut que la projection se fasse sur `R` et pas sur

`T_metric rn R_Met`. D'ailleurs si nous avons déplié certaines définitions, c'était dans le seul but de remplacer `T_metric rn R_Met` par `R`. Nous aimerions pouvoir faire la même chose, mais sans avoir à déplier des définitions, c'est à dire que l'on voudrait une fonction qui ait un comportement semblable à `EspMet_Top` sauf que l'on pourrait lui passer en argument le type sur lequel on veut que la projection soit enregistrée (i.e. le type `R` des réels de Coq dans notre exemple). Pour cela il faut séparer le type de base des autres champs de la structure. Nous définissons donc une première structure que nous allons appeler `mixin_of` pour laquelle le type de base sera en paramètre :

```
Structure mixin_of (T_Space : Type) : Type := Mixin
 { open : (T_Space -> Prop) -> Prop;
   _ : open ...
   _ : ...
            ⋮
 }.
```

Cette structure ne suffit pas, car comme le type de base n'est plus un champ de la structure, lors de la déclaration de structure canonique, la projection vers le type de base n'existera pas. Les projections enregistrées correspondent seulement aux champs de la structure qui sont nommés. Nous allons donc définir une deuxième structure qui va correspondre à notre type final d'espace topologique (ou métrique) :

```
Notation class_of := mixin_of
Structure type := Pack {sort : Type; _ : class_of sort}.
Coercion sort : type >-> Sortclass.
```

La structure `class_of` est habituellement utilisée pour assurer l'héridité dans les structures héréditaires. Comme ici nous n'avons pas d'hérédité, `class_of` est seulement une notation utilisée pour des raisons d'uniformité des définitions.

Ces structures sont définies dans un module, par exemple ici dans le module `Topology`. Ce qui fait qu'en dehors du module les noms des structures seront de la forme `Topology.type` ce qui permet d'avoir des noms uniformes. Chacune de ces nouvelles définitions sera utilisée avec les notations suivantes :

```
Notation topologyType := type.
Notation TopologyMixin := Mixin.
Notation TopologyType T m := (@Pack T m).
```

Nous avons donc séparé le type de base et les axiomes de la structure que nous avons mis dans une boîte que nous avons appelée `mixin_of`. Cette boîte a été mise dans une deuxième structure appelée `type` qui contient comme premier champ le type sur lequel nous voulons déclarer l'instance, et comme deuxième champ la boîte qui contient toutes les propriétés que doit vérifier le type en question.

Pour revenir au problème qui nous intéresse, avec ces nouvelles structures nous allons déclarer l'instance canonique d'espace topologique à partir d'un espace métrique :

```
Variable Rn : numDomainType.
Variable E : (espMetType Rn).

Definition EspMet_TopMixin := Eval hnf in
  TopologyMixin empty_in_open_met full_in_open_met ...
Definition EspMet_TopType := Eval hnf in
  TopologyType E EspMet_TopMixin.

Canonical EspMet_TopMixin.
Canonical EspMet_TopType.
```

Nous déclarons ensuite de la même manière que l'ensemble des nombres réels de Coq (représenté par `R`) a une structure d'espace métrique :

```
Definition EspMet_RMixin := EspMetMixin R_dist_pos ...
Definition EspMet_R :=
 Eval hnf in EspMetType R R EspMet_RMixin.
      ⋮
```

Et maintenant pour dire que le type `R` des réels de Coq a une structure d'espace topologique, nous allons utiliser la fonction `TopologyType`. Comme nous voulons que la projection se fasse directement sur `R`, nous allons donner comme premier argument `R`. Le deuxième argument doit être la structure `mixin_of`, la notation `EspMet_TopMixin` permet justement de construire cette structure. Nous pouvons donc déclarer la structure d'espace topologique sur `R` directement comme ceci :

```
Definition EspTop_R := TopologyType R (EspMet_TopMixin EspMet_R).
Canonical EspTop_R.
```

Bien que dans cette sous-section nous mentionnons les espaces normés et euclidiens (avec un produit scalaire), seules les structures d'espaces métriques et topologiques sont implémentées. La formalisation des espaces normés et euclidiens pourra faire l'objet de travaux futurs. Nous allons maintenant nous intéresser à la formalisation de certains concepts de topologie générale qui ont permis d'énoncer et de prouver formellement le théorème de Bolzano-Weierstraß.

3.3 Bolzano-Weierstraß

Le théorème de Bolzano-Weierstraß est un résultat de topologie générale dont la forme la plus utilisée est la suivante :

Bolzano-Weierstraß (dans un espace métrique)**.** *Toute suite à valeurs dans un compact admet une sous-suite convergente.*

La version du théorème formalisée est sa forme plus générale qui est vraie pour un espace topologique quelconque :

Bolzano-Weierstraß (dans un espace topologique)**.** *Toute suite à valeurs dans un compact admet une valeur d'adhérence.*

En effet, dans un espace métrique les valeurs d'adhérence sont les limites de sous-suites.

Nous donnons dans un premier temps quelques définitions de topologie générale, principalement celles dont nous aurons besoin (comme la compacité par exemple). Nous verrons ensuite quelques notions sur les suites, et enfin nous parlerons de la formalisation du théorème.

3.3.1 Topologie générale

Nous faisons ici quelques rappels des notions utilisées par la suite.

Une première notion que nous utiliserons est celle de voisinage. Soit x un point d'un espace topologique $\mathcal{T}$, alors un ensemble V est un voisinage du point x si V contient un ouvert qui contient x, autrement dit s'il existe un ensemble ouvert qui contient x et qui est inclus dans V :

```
Definition neighbourhood (V : T -> Prop) (x : T) :=
  exists A, [/\ x 'in A, open A & A <s V].
```

Si pour deux points distincts d'un espace topologique on peut trouver deux voisinages disjoints de chacun de ces points, alors on dit que l'espace topologique est séparé :

```
Definition separated := forall x y, x <> y -> exists V1, exists V2,
[/\ neighbourhood V1 x, neighbourhood V2 y & (inter V1 V2) <s {ø : T}].
```

où `inter V1 V2` est l'intersection des ensembles `V1` et `V2`.

Nous définissons maintenant la notion d'ensemble compact, utile au théorème de Bolzano-Weierstraß. La propriété qui nous intéresse est le fait que, si nous avons un ensemble K et une famille d'ouverts f qui recouvre K, alors on peut trouver une sous-famille finie de f qui recouvre K. Un ensemble qui vérifie cette propriété est dit quasi-compact, on parlera de compacité dans le cas où nous sommes dans un espace topologique séparé :

```
Definition quasi_compact (K : T -> Prop) :=
  forall Ti (f : family T Ti),
  open_family f -> cover K f ->
  exists g : finite_family T Ti, sub_family f g /\ cover K g.

Definition compact (K : T -> Prop) := separated /\ quasi_compact K.
```

Nous allons maintenant définir les suites.

3.3.2 Les suites

Nous parlerons ici de sous-suites et nous rappellerons en donnant les définitions formelles ce qu'est une valeur d'adhérence et la convergence d'une suite.

Une suite d'éléments de type `T` est représentée par un élément dont le type est `nat -> T`. Soit $u = (u_n)_{n\in\mathbb{N}}$ une suite, alors toutes les sous-suites de u peuvent

s'exprimer sous la forme $(u_{\varphi(n)})_{n\in\mathbb{N}}$ où φ est une fonction strictement croissante de $\mathbb{N} \to \mathbb{N}$. Donc une sous-suite est principalement la donnée d'une fonction φ qui vérifie les bonnes propriétés :

```
Structure sub_seq : Type := mkss
  {
    phi :> nat -> nat;
    grow : forall m n, (m < n)%N -> (phi m < phi n)%N
  }.
```

Formellement, étant donné `g : sub_seq` et une suite `Un : nat -> T`, alors `Un \o g` (où `\o` est la composition) désigne une sous-suite de `Un`.

La formalisation des définitions de convergence et de valeur d'adhérence est une retranscription directe des définitions mathématiques. Une suite $(u_n)_{n\in\mathbb{N}}$ converge vers une limite l si pour tout voisinage V de l on peut trouver un rang à partir duquel tous les éléments de la suite sont dans V. Et l est une valeur d'adhérence si pour tout voisinage V de l, pour chaque rang N on peut trouver un élément de la suite dans V au-delà du rang N :

```
Definition Un_cvg (Un : nat -> T) (l : T) :=
  forall V, neighbourhood V l ->
  exists N : nat, forall n, N <= n -> (Un n) 'in V.
Notation "Un >->> l" := (Un_cvg Un l).

Definition cluster_point (Un : nat -> T) (l : T) :=
  forall V, neighbourhood V l ->
  forall N : nat, exists n, N <= n /\ (Un n) 'in V.
```

À ce point, nous avons toutes les définitions utiles à l'énoncé et à la preuve du théorème de Bolzano-Weierstraß, dont nous décrivons la formalisation dans la section suivante.

3.3.3 Le théorème

Le théorème de Bolzano-Weierstraß s'énonce formellement sur un espace topologique quelconque de la manière suivante :

```
Lemma Bolzano_Weierstrass : forall (Un : nat -> T) K, compact K ->
  (forall n, (Un n) 'in K) -> exists l, l 'in (cluster_point Un).
```

Nous allons dans un premier temps voir les grandes lignes de la preuve de ce théorème.

Pour utiliser le théorème sous sa forme plus usuelle dans un espace métrique, nous montrerons ensuite que dans un espace métrique, les valeurs d'adhérence d'une suite sont des limites de ses sous-suites.

Preuve dans un espace topologique

La preuve formelle du théorème suit les mêmes étapes que la démonstration mathématique, et sa formalisation n'a pas posé de problème majeur. Les grands

points de la démonstration du théorème sont les suivants :

Soit K un ensemble compact et $u = (u_n)_{n\in\mathbb{N}}$ une suite dont tous les termes sont dans l'ensemble K. Supposons par l'absurde que la suite u n'ait aucune valeur d'adhérence. Cela signifie que pour n'importe quel point $x \in K$ on peut trouver un voisinage (ouvert) V_x de x qui à partir d'un certain rang n_x ne contient aucun élément de la suite. La famille $(V_x)_{x\in K}$ recouvre l'ensemble K, comme K est compact, alors on peut extraire une sous-famille finie $(W_i)_{i\in\{0...k\}}$ avec $W_i = V_{x_i}$ qui recouvre K. Pour chaque W_i on a un rang n_{x_i} au-delà duquel la suite u n'a aucun terme qui appartient à l'ensemble W_i. Donc si l'on note N le maximum de tous les n_{x_i}, u_N n'appartient à aucun ensemble de la famille W. Ce qui est absurde car u_N est dans l'ensemble K et que la famille W forme un recouvrement de K.

Preuve dans un espace métrique

Pour montrer la version métrique du théorème de Bolzano-Weierstraß, nous montrons que dans les epaces métriques, les valeurs d'adhérence d'une suite sont les limites de ses sous-suites.

Dans un espace métrique, nous avons une distance, et cela nous permet de définir pour un point x et un réel positif r donnés, la boule de centre x et de rayon r, notée généralement $\mathcal{B}(x, r)$, dont nous rappelons la définition formelle :

```
Definition open_ball (x : E) (r : {posD Rn}) (y : E) := d x y < r.
Notation "''B' ( x , r )" := (open_ball x r).
```

où `d` est la distance de l'espace métrique.

Ainsi, pour la plupart des définitions comme la convergence ou les valeurs d'adhérence d'une suite, on peut remplacer de manière équivalente les voisinages par des boules.

Comme nous l'avons vu précédemment, les définitions formelles sont principalement de simples retranscriptions des définitions mathématiques. Donc dans la suite, nous utiliserons pour plus de lisibilité les notations mathématiques.

Soit donc $u = (u_n)_{n\in\mathbb{N}}$ une suite, et l une valeur d'adhérence de u. Le fait d'enlever le premier terme d'une suite ne change pas les valeurs d'adhérence, donc l est aussi une valeur d'adhérence de la suite $(u_{n+1})_{n\in\mathbb{N}}$, autrement dit :

$$\forall V \in \mathcal{V}\text{ois}(l), \forall N, \exists n, N \leq n \text{ et } u_{n+1} \in V$$

où $\mathcal{V}\text{ois}(l)$ désigne l'ensemble des voisinages de l. Comme nous l'avons dit plus haut, ceci est équivalent à :

$$\forall k \in \mathbb{N}^*, \forall N, \exists n, N \leq n \text{ et } u_{n+1} \in \mathcal{B}(l, \tfrac{1}{k}) \tag{1}$$

Nous voulons maintenant montrer que la suite u a une sous-suite qui converge vers l, autrement dit nous voulons trouver une fonction $\varphi : \mathbb{N} \to \mathbb{N}$ strictement croissante telle que la suite $(u_{\varphi(n)})_{n\in\mathbb{N}}$ converge vers l. Nous allons donc dans un premier temps "construire" la fonction φ.

Pour construire la fonction, nous allons utiliser le lemme `functional_choice` que l'on trouve dans la bibliothèque standard de Coq[1]. Ce lemme s'énonce ainsi :

$$(\forall x \in A, \exists y \in B, P(x,y)) \Leftrightarrow (\exists f : A \to B, \forall x \in A, P(x, f(x)))$$

Si nous appliquons ce lemme une première fois dans l'expression (1) nous obtenons :

$$\forall k \in \mathbb{N}^*, \exists f, \forall N, N \leq f(N) \text{ et } u_{f(N)+1} \in \mathcal{B}(l, \tfrac{1}{k})$$

Nous pouvons utiliser le lemme `functional_choice` une seconde fois pour obtenir :

$$\exists G, \forall k \in \mathbb{N}^*, \forall N \in \mathbb{N}, N \leq G_k(N) \text{ et } u_{G_k(N)+1} \in \mathcal{B}(l, \tfrac{1}{k})$$

où $G_k(N)$ est une notation pour $(G(k))(N)$.

Nous avons donc une fonction G du type `nat->nat->nat` qui vérifie :

$$\forall k \in \mathbb{N}^*, \forall N \in \mathbb{N}, N \leq G_k(N) \text{ et } u_{G_k(N)+1} \in \mathcal{B}(l, \tfrac{1}{k})$$

ou de manière équivalente :

$$\forall k \in \mathbb{N}^*, \forall N \in \mathbb{N}, N < G_k(N) + 1 \text{ et } u_{G_k(N)+1} \in \mathcal{B}(l, \tfrac{1}{k}) \tag{2}$$

Ici nous avons fait apparaître deux choses. La première est l'inégalité $N < G_k(N) + 1$ et la deuxième est $u_{G_k(N)+1}$ qui est un bon candidat pour définir une sous-suite. Le problème c'est que l'expression $G_k(N) + 1$ dépend de deux entiers k et N. Mais l'expression (2) est vraie quelque soit N, donc en particulier elle est vraie pour tous les N de la forme $\varphi(k)$ pour une certaine fonction φ. Ainsi en remplaçant dans l'inégalité les occurences de N par $\varphi(k)$, où φ est une fonction que l'on cherche à définir, nous obtenons d'une part :

$$\forall k, \varphi(k) < G_k(\varphi(k)) + 1$$

Donc en définissant récursivement la fonction φ de manière à ce que $\varphi(k+1) = G_k(\varphi(k)) + 1$, nous avons $\forall k, \varphi(k) < \varphi(k+1)$ et donc φ est une fonction strictement croissante. Et d'autre part, d'après (2) la fonction φ vérifie :

$$\forall k \in \mathbb{N}^*, u_{\varphi(k+1)} \in \mathcal{B}(l, \tfrac{1}{k}) \tag{3}$$

On peut définir la fonction φ formellement comme suit :

```
Fixpoint phi (m : nat) : nat :=
  match m with
    | 0 => G 0 0
    | m'.+1 => (G m' (phi m')).+1
  end.
```

[1] http://coq.inria.fr/distrib/current/stdlib/Coq.Logic.IndefiniteDescription.html

Nous avons donc construit une fonction φ strictement croissante. Maintenant, il ne nous reste plus qu'à prouver que la suite $u_{\varphi(n)}$ converge vers l, ainsi nous aurons montré que l est une limite d'une sous-suite de u.

Comme nous l'avons déjà dit précédemment, on peut remplacer les voisinages par des boules de manière équivalente dans la définition de convergence dans un espace métrique. Nous voulons donc montrer que :

$$\forall t \in \mathbb{N}^*, \exists N \in \mathbb{N}, \forall n, N \leq n \Rightarrow u_{\varphi(n)} \in \mathcal{B}(l, \tfrac{1}{t})$$

Soit $t \in \mathbb{N}^*$, alors montrons que $N = t + 1$ convient. Soit $n \in \mathbb{N}$ tel que $t + 1 \leq n$, il nous reste à montrer que $u_{\varphi(n)} \in \mathcal{B}(l, \frac{1}{t})$.

Comme $t+1 \leq n$ et que $t \in \mathbb{N}^*$ alors $n-1 \in \mathbb{N}^*$, donc d'après (3) nous avons :

$$u_{\varphi(n)} \in \mathcal{B}(l, \tfrac{1}{n-1})$$

Comme $t + 1 \leq n$ alors $t \leq n - 1$, ce qui implique que $\frac{1}{n-1} \leq \frac{1}{t}$, et donc $\mathcal{B}(l, \frac{1}{n-1}) \subset \mathcal{B}(l, \frac{1}{t})$, autrement dit :

$$u_{\varphi(n)} \in \mathcal{B}(l, \tfrac{1}{t})$$

Nous avons donc montré que dans un espace métrique, les valeurs d'adhérences d'une suite sont les limites de ses sous-suites. La preuve du résultat précédent permet d'utiliser la version de Bolzano-Weierstraß dans un espace métrique.

En fait, dans un cadre plus général, pour montrer ce résultat il suffit d'avoir pour chaque point l de l'espace topologique une base dénombrable de voisinages (dans les espaces métriques ce sont les boules de centre l et de rayon $\frac{1}{k}$).

3.4 Conclusion

La topologie générale se formalise plutôt bien. Comme nous l'avons vu, la plupart des définitions sont naïves, et les preuves suivent le même schéma que les preuves mathématiques. La logique utilisée est la logique classique. D'une part cela permet d'avoir une théorie des ensembles avec les bonnes propriétés (comme le fait qu'un ensemble soit égal au complémentaire de son complémentaire par exemple), d'autre part le théorème de Bolzano-Weierstraß est un résultat de logique classique.

La formalisation de la topologie a permis de prouver le théorème de Bolzano-Weierstraß, mais aussi de munir les structures ordonnées de la bibliothèque SSReflect d'une topologie, et de montrer des résultats dessus dont nous aurons également besoin par la suite.

Il existe déjà un théorème de Bolzano-Weierstraß dans la bibliothèque standard de Coq, mais prouvé uniquement sur les nombres réels. Puisque nous l'avons prouvé dans un cadre plus général, cela fait que l'on pourra l'utiliser sur n'importe quel type que l'on aura muni d'une topologie. En particulier, par la suite, nous aurons besoins de l'utiliser sur l'espace des matrices.

Cependant, pour utiliser ce théorème il faut d'abord montrer qu'un certain ensemble est compact. L'argument souvent utilisé pour prouver la compacité d'un ensemble est de montrer qu'il est fermé et borné. Mais pour prouver ce résultat il faut être dans un espace vectoriel normé de dimension finie ou dans un espace euclidien. Comme nous l'avons vu, pour l'instant seules les structures d'espaces métriques et topologiques sont implémentées, il serait donc intéressant par la suite d'implémenter également les structures d'espaces normés et euclidiens.

3.5 Travaux reliés

Les travaux de formalisation de topologie générale sont nombreux et ont été faits dans de nombreux assistants de preuve comme Mizar [38], Isabelle [26], PVS [30], etc. Plus particulièrmeent dans l'assistant de preuve HOL Light on trouve une preuve du théorème de Bolzano-Weierstraß dans le cas particulier des espaces métriques [23].

Il existe également un début de formalisation de topologie générale en Coq, dans un projet qui s'appelle Coqtail[1].

Il existe aussi des travaux de formalisation qui ont une approche constructive de la topologie. Cela la s'appelle la topologie formelle, ou la topologie sans points. On pourra par exemple cité les travaux de Giovanni Sambin [43] dont une partie des travaux a été formalisée avec l'assistant de preuve Matita [2].

[1] http://coqtail.sourceforge.net

Chapitre 4

Perron-Frobenius

Le théorème de Perron-Frobenius est un théorème d'algèbre linéaire à propos du rayon spectral d'une matrice réelle dont les coefficients sont positifs. Les principales applications de ce théorème sont en théorie des graphes, ou en probabilité avec les chaînes de Markov. Le théorème est utilisé sur les matrices d'adjacence d'un graphe ou sur des matrices stochastiques, ce sont des matrices à coefficients positifs [48]. On utilise également le théorème de Perron-Frobenius dans l'étude des matrices dont les coefficients sont des intervalles [32, 42].

La partie du théorème de Perron-Frobenius à laquelle on s'intéresse est celle qui dit que le rayon spectral d'une matrice à coefficients positifs est une valeur propre de la matrice, et que cette valeur propre a un vecteur propre associé dont les coefficients sont positifs.

Cette preuve se fait en deux étapes. Dans une première étape on démontre le théorème pour les matrices à coefficients strictement positifs. Dans une deuxième étape on utilise un argument de densité pour prouver le résultat sur les matrices à coefficients positifs ou nuls. Pour prouver ce théorème nous devrons utiliser les notions de topologie, et la forme normale de Jordan d'une matrice vues dans les deux chapitres précédents.

Dans un premier temps, nous allons voir dans ce chapitre la formalisation de la première étape de la preuve. Nous verrons d'abord les définitions de base dont nous aurons besoin par la suite. Ensuite nous présenterons une preuve du théorème pour le cas des matrices strictements positives pour mettre en évidence les résultats intermédiaires dont nous aurons besoin et dont nous montrerons la formalisation ensuite.

Dans un second temps, nous expliquerons le chemin qu'il reste à faire pour la formalisation de la deuxième partie de la preuve.

4.1 Rappels et définitions

4.1.1 Minimum et maximum

Nous aurons besoin de définir le maximum d'une famille d'éléments pour la définition du rayon spectral d'une matrice et du minimum pour la preuve du théorème. C'est pourquoi nous commençons par définir ces deux notions dans cette section.

Minimum

Pour exprimer le minimum d'une famille d'éléments, on pourrait essayer d'utiliser les opérateurs de familles de la façon suivante[1] :

```
\big[minr/id]_(i <- s) F i
```

Le problème avec cette façon de faire c'est que `id` doit être un élément neutre pour le minimum. En effet, nous avons vu que, par exemple, pour une séquence `[:: a, b, c]` l'expression ci-dessus est équivalente à :

```
minr a (minr b (minr c id))
```

et donc si `a`, `b` et `c` sont tous plus grand que l'élément `id` alors la valeur retournée sera `id` et non le plus petit des trois éléments `a`, `b` et `c`. C'est la raison pour laquelle `id` doit être un élément neutre, or dans notre cas la fonction minimum n'a pas d'élément neutre.

Pour les besoins de la preuve, nous avons seulement besoin d'une fonction qui retourne un élément de la famille qui soit plus petit que tous les autres. Le minimum est donc simplement défini comme suit :

```
Fixpoint min_seq s :=
  match s with
   | [::] => 0
   | [:: x] => x
   | x :: s' => minr x (min_seq s')
  end.
```

On traite à part le cas où la séquence ne contient qu'un seul élément pour que l'appel récursif de la fonction ne se fasse jamais sur une séquence vide. À partir de cette définition, nous pouvons définir une fonction minimum plus proche de ce qui est fait dans la bibliothèque SSReflect sur les opérateurs de familles :

```
Definition min_fT (I : finType) (F : I -> R) :=
  min_seq [seq F i | i <- index_enum I].
```

Cette définition respecte les deux propriétés dont nous avons besoin, à savoir :

```
Lemma min_fT_le (I : finType) F (i : I) : min_fT F <= F i.
Lemma ex_min_fT (I : finType) F (_ : I) : {k | min_fT F = F k}.
```

[1] voir 1.2.5 et 1.2.10 pour les notations.

Maximum

Pour définir le maximum d'une famille d'éléments, nous pouvons contourner le problème de l'élément neutre du fait que dans notre contexte, nous prenons toujours le maximum d'une famille d'éléments positifs. Nous pouvons donc choisir zéro comme élément neutre et utiliser les opérateurs de familles :

```
\big[(@maxr R)/0%R]_i F i
```

L'expression ci-dessus sera cachée derrière la notation :

```
\maxr_i F i
```

4.1.2 Éléments d'algèbre linéaire

Soit A une matrice carrée de dimension n à coefficients dans un corps. Le noyau de la matrice A, noté $\ker A$, est l'ensemble des vecteurs tels que $Ax = 0$ c'est-à-dire :

$$x \in \ker A \Leftrightarrow Ax = 0$$

Le noyau d'une matrice est fourni par une fonction de la bibliothèque SSReflect, noté `kermx A`.

Si x est un vecteur non nul qui appartient au noyau de la matrice A, alors A n'est pas inversible. En effet, si A était inversible nous aurions :

$$x = I * x = (A^{-1} * A)x = A^{-1}(Ax) = A^{-1} * 0 = 0$$

ce qui contredit le fait que x soit non nul. La matrice A n'étant pas inversible, comme nous sommes sur des matrices à coefficients dans un corps, nous avons $\det A = 0$.

Soit maintenant x et λ respectivement un vecteur colonne non nul et un scalaire tels que l'on ait la relation :

$$Ax = \lambda x$$

Alors on dit que λ est une valeur propre de la matrice A et que x est un vecteur propre associé à la valeur propre λ.

La relation ci-dessus peut être réécrite comme suit :

$$(A - \lambda I)x = 0 \tag{1}$$

Donc si λ est une valeur propre de la matrice A alors x est un vecteur non nul appartenant au noyau de la matrice $(A - \lambda I)$. C'est ainsi qu'est définie le prédicat booléen qui caractérise les valeurs propres dans la bibliothèque SSReflect :

```
Variables (T : fieldType) (n : nat) (A : 'M[T]_n).
Definition eigenvalue lam := kermx (A - lam%:M) != 0.
```

Remarque 2. *Le prédicat* `eigenvalue` *ne représente que l'ensemble des valeurs propres qui sont de type* `T`*. Donc par exemple si* `T` *est le corps des réels, alors les seules valeurs qui vérifieront le prédicat seront les valeurs propres réelles de la matrice A, et si* `lam` *est une valeur propre complexe de la matrice A alors* `eigenvalue A lam` *sera mal typé.*

L'égalité (1) indique également que si λ est une valeur propre de la matrice A alors $\det(A - \lambda I) = 0$ d'après ce que l'on a vu juste avant. C'est-à-dire que λ est une racine du polynôme caractéristique de la matrice A. Nous pouvons donc, dans un corps algébriquement clos, définir la séquence des valeurs propres d'une matrice :

```
Variable F : closedFieldType.
Definition eigen_seq n (A : 'M[F]_n) := root_seq (char_poly A).
```

où la fonction `root_seq` retourne la séquence des racines d'un polynôme. Par exemple, si `p` est le polynôme `('X-1)^+2 * ('X-2)` alors `root_seq p` sera la séquence `[:: 1, 1, 2]` (ou une de ses permutations).

Le lemme suivant montre que, sur un corps algébriquement clos, les deux définitions ci-dessus expriment bien la même chose :

```
Lemma eigenE n (A : 'M[F]_n) lam :
  (lam \in (eigenvalue A)) = (lam \in (eigen_seq A)).
```

Le rayon spectral d'une matrice est le plus grand des modules des valeurs propres d'une matrice. Donc pour définir le rayon spectral, il faut une fonction module. Comme la séquence des valeurs propres est définie pour les matrices à coefficients dans un corps clos, alors le rayon spectral est défini pour les matrices à coefficients complexes[1] :

```
Variable R : rcfType.
Notation C := (complex R).

Definition spectral_radius n (A : 'M[C]_n) :=
  \maxr_(lam <- eigen_seq A) Re `|lam|.
Notation "\rho A" := (spectral_radius A).
```

où `Re` désigne la fonction partie réelle, car bien que le module d'un nombre complexe ait une partie imaginaire nulle, son type est le type des nombres complexes, donc la fonction partie réelle est utilisée ici pour que le résultat obtenu soit réel. Donc pour parler du rayon spectral d'une matrice à coefficients réels, il faudra d'abord plonger la matrice dans l'espace des matrices à coefficients complexes.

Dans la suite, nous parlerons également de comparaison entre matrices. Si A et B sont deux matrices de même dimensions, alors on dira que $A \leq B$ (resp. $A < B$) si tous les coefficients de B sont supérieurs ou égaux (resp. strictement supérieurs) aux coefficients de A. Dans le code formel, nous utiliserons les définitions et les notations suivantes :

[1] voir 1.2.10 et 1.2.11 pour les notations.

```
Definition Mle A B := forall i j, A i j <= B i j.
Definition Mlt A B := forall i j, A i j < B i j.

Notation "A <=m: B" := (Mle A B).
Notation "A <m: B" := (Mlt A B).
```

Enfin, si A est une matrice alors $|A|$ désigne la matrice A à laquelle on a appliqué la fonction valeur absolue (ou module pour une matrice complexe) à chacun de ses coefficients. Ceci est défini comme suit :

```
Definition Mabs A := \matrix_(i,j) `|A i j|.
```

Les trois définitions précédentes sont reprises des travaux de Ioana Paşca [42]. Avec ces définitions, le théorème de Perron-Frobenius s'énonce ainsi :

Perron-Frobenius Soit $A \in \mathcal{M}_n(\mathbb{R})$ telle que $0 \leq A$, alors $\rho(A)$ est une valeur propre de la matrice A et possède un vecteur propre associé dont tous les coefficients sont positifs.

Avec les définitions formelles que nous avons vues jusqu'à présent, le théorème s'énoncerait ainsi :

```
Lemma Perron-Frobenius n (A : 'M[R]_n) : 0 <=m: A ->
  exists x, 0 <=m: x /\ A *m x = (\rho A^%:C) *: x.
```

La notation `A^%:C` signifie que chaque coefficient réel de la matrice `A` est plongé dans le corps des nombres complexes

4.2 Matrices strictement positives

Nous allons voir maintenant dans une première partie la formalisation de la preuve du théorème de Perron-Frobenius pour les matrices strictement positives. Dans une seconde partie, nous verrons la formalisation d'un résultat important dont nous aurons besoin pour la première partie.

4.2.1 Preuve du théorème principal

Le théorème que nous allons montrer ici est le suivant :

Perron-Frobenius[cas strictement positif] Soit $A \in \mathcal{M}_n(\mathbb{R})$ telle que $0 < A$, alors $\rho(A)$ est une valeur propre de la matrice A et possède un vecteur propre associé dont tous les coefficients sont strictement positifs.

La preuve du théorème repose sur l'axiome suivant :

Axiome. *Soit $A \in \mathcal{M}_n(\mathbb{R})$. La suite définit par $(A^k)_{k\in\mathbb{N}}$ converge vers 0 si et seulement si $\rho(A) < 1$.*

Nous utiliserons également dans la preuve le résultat suivant :

Resultat 1. *Soit $A \in \mathcal{M}_n(\mathbb{R})$ telle que $0 < A$, soit x un vecteur non nul tel que $0 \leq x$, alors $0 < Ax$.*

En effet, le i-ème coefficient du vecteur Ax est $\sum_k a_{ik}x_i$ et comme x est non-nul alors il existe un terme de la somme qui est non-nul, donc tous les coefficients du vecteur Ax sont strictement positifs.

Pour la preuve du théorème de Perron-Frobeniuspour les matrices strictement positives nous avons besoins du lemme suivant :

Lemme 5. *Soit $A \in \mathcal{M}_n(\mathbb{R})$ telle que $0 < A$. Soit λ une valeur propre de A telle que $|\lambda| = \rho(A)$, et soit x un vecteur non nul tel que $Ax = \lambda x$. Alors nous avons $A|x| = \rho(A)|x|$, $0 < \rho(A)$ et $0 < |x|$.*

Pour montrer ce lemme, on fait un traitement par cas sur l'égalité $A|x| = \rho(A)|x|$:

Posons $z = A|x|$, alors d'après le résultat 1 nous avons $0 < z$.

- Supposons que $A|x| = \rho(A)|x|$, dans ce cas il ne reste plus qu'à montrer que $0 < \rho(A)$ et $0 < |x|$. On sait que $0 \leq \rho(A)$, si $\rho(A) = 0$ alors on aurait $A|x| = 0 * |x| = 0$, autrement dit on aurait $z = 0$ ce qui contredit le fait que $0 < z$, donc $0 < \rho(A)$. De plus, comme $0 < z$ et que $z = \rho(A)|x|$, on a $0 < \rho(A)|x|$ et donc $0 < |x|$.

- Maintenant supposons que $A|x| \neq \rho(A)|x|$. Nous allons montrer que cette hypothèse est absurde. Posons $y = A|x| - \rho(A)|x|$, alors d'après l'inégalité précédente nous avons $y \neq 0$, de plus $0 \leq y$ en effet, nous avons :

  $$\rho(A)|x| = |\lambda||x| = |\lambda x| = |Ax| \leq |A||x| = A|x|$$

  et donc
  $$\rho(A)|x| \leq A|x| \Leftrightarrow 0 \leq A|x| - \rho(A)|x| = y$$

  Donc d'après le résultat 1 nous avons $0 < Ay = Az - \rho(A)z$.

  Comme $0 < z$ et $0 < Az - \rho(A)z$ alors il existe un $\varepsilon > 0$ tel que l'on ait $Az - \rho(A)z > \varepsilon z$. Autrement dit, nous avons :

  $$Az > (\rho(A) + \varepsilon)z \Leftrightarrow \frac{A}{\rho(A) + \varepsilon} z > z$$

  Donc en posant $B = A/(\rho(A) + \varepsilon)$ nous avons $Bz > z$, et par récurrence nous obtenons que $\forall k, B^k z > z$. De plus $\rho(B) = \rho(A)/(\rho(A) + \varepsilon) < 1$ donc d'après l'axiome, la suite des B^k converge vers 0 ce qui donne que $0 \geq z > 0$ ce qui est absurde.

Le théorème découle directement du lemme 5 et des deux lemmes ci-dessous :

```
Lemma ex_eigen_rho n (A : 'M[C]_n.+1) :
  {lam | eigenvalue A lam & `|lam| = \rho(A)%:C}.

Lemma eigenvalueP a :
  reflect (exists2 v : 'rV_n, v *m g = a *: v & v != 0) (eigenvalue a).
```

Le premier est une conséquence du fait que le maximum d'un nombre fini de valeurs est atteint et le deuxième est un résultat donné par la bibliothèque SS-Reflect.

Dans la preuve du lemme 5, nous avons utilisé quatre résultats qui ne sont pas seulement des manipulations algébriques :

- Le premier dit que si u et v sont des vecteurs strictement positifs, alors il existe $\varepsilon > 0$ tel que $\varepsilon v < u$. Pour montrer ce résultat, on peut prendre $\varepsilon = \min_i \frac{u_i}{2v_i}$. L'énoncé prouvé formellement est celui-ci :

```
Lemma Mle_eps m (u v : 'cV[R]_m.+1) :
    let F i := ((u i 0) / 2%:R)/ (v i 0) in
    let eps := min_fT F in
    0 <m: u -> 0 <m: v -> eps *: v <m: u.
```

- Le deuxième dit que $\rho(kA) = |k|\rho(A)$ (nous l'utilisons pour montrer que $\rho(B) < 1$) ceci vient du fait que si $Ax = \lambda x$ alors $kAx = k\lambda x$ donc les valeurs propres de kA sont k fois les valeurs propres de A, donc quand on prend le maximum des modules des valeurs propres, il y a $|k|$ qui apparaît en facteur. Ce résultat s'énonce formellement comme suit :

```
Lemma rho_scalemx n a (A :'M[C]_n) :
  \rho (a *: A) = Re `|a| * \rho A.
```

- Le troisième résultat utilisé est que si $\forall k, A_k > B$, et si la suite des A_k converge vers C alors on a $C \geq B$. Son énoncé formel est le suivant :

```
Lemma ltmx_cvg_lemx m n (A B :'M[R]_(m,n)) (U : nat -> 'M_(m,n)) :
  (forall k, A <m: U k) -> (U >->> B) -> A <=m: B.
```

Nous remarquons que certaines propriétés utilisent la notion de convergence d'une suite de matrices. Donc pour pouvoir les exprimer, il faut d'abord munir le type des matrices d'une topologie. Nous allons donc voir maintenant comment est implémentée la structure d'espace topologique sur les matrices, mais aussi sur les structures ordonnées car nous en aurons besoin par la suite.

4.2.2 Instance des structures topologiques

Nous avons déjà vu au chapitre 3 comment sont implémentées les structures d'espace métrique et topologique. Nous avons aussi vu sur l'exemple des réels de Coq comment instancier ces structures. Aussi nous indiquerons ici seulement la métrique qui sera utilisée pour ces instanciations.

Les structures ordonnées

Toutes la hiérarchie des structures ordonnées est basée sur la structure appelée `numDomainType`. Nous avons vu que cette structure possède une fonction *valeur absolue*, ou *module* notée `` `|.| ``. La métrique utilisée sur les structures ordonnées est donc la suivante :

```
Definition dist_real x y := `|x - y|.
```

Les nombres complexes

Nous avons vu aussi que dans la définition d'espace métrique le type de retour de la fonction distance était en paramètre. Ainsi si nous avons `E : metricType R` alors plus `R` sera "haut" dans la hiérarchie des structures ordonnées, plus nous pourrons démontrer de propriétés sur les espaces métriques. Les nombres complexes sont une instance de la structure `numFieldType`. Or nous aurons besoin dans le développement formel que le type de retour de la fonction distance sur les nombres complexes soit au moins de type `realDomainType`. C'est pourquoi nous utilisons la fontion partie réelle dans la définition de la distance sur les nombres complexes :

```
Definition dist_C x y := Re `|x - y|.
```

Les matrices

Avant de définir une distance sur les matrices, nous définissons d'abord un type pour les normes de matrices :

```
Section normDef.

Variable R : numDomainType.

Structure can_norm : Type := MNorm {
  anorm : forall p q, 'M[R]_(p,q) -> R;
  pos_norm : forall p q (A : 'M_(p, q)), 0 <= anorm A;
  hom_pos_norm : forall p q (A : 'M_(p, q)) (r : R),
    anorm (r *: A) = `|r| * anorm A;
  trin_ineq_norm : forall p q (A B : 'M_(p, q)),
    anorm (A + B) <= anorm A + anorm B;
  prod_ineq_norm : forall p q r (A : 'M_(p, q)) (B : matrix _ q r),
    anorm (A *m B) <= anorm A * anorm B;
  canonical_norm : forall p q (A : 'M_(p, q) ) i j,
     `|A i j| <= anorm A;
  canonical_norm2 : forall p q (A B: 'M_(p, q)),
    (forall i j, `|A i j| <= `|B i j|) -> anorm A <= anorm B
}.
End normDef.
```

```
Notation "|| A : N |" := (anorm N A).
```

Ici nous avons simplement repris la définition de norme formalisée par Ioana Paşca dans [42]. La définition a été actualisée avec les nouvelles définitions et notations de SSReflect.

La norme que nous utiliserons par la suite est la norme *infini* définie commme suit[1] :

```
Definition norm8 (m n : nat) (A : 'M[R]_(m,n)) :=
  \maxr_i \sum_j `|A i j|.
```

Cette norme vérifie tous les axiomes de la structure `can_norm` définie ci-dessus. Donc après avoir démontré ces propriétés, nous pouvons déclarer la norme *infini* comme une instance de la structure :

```
Canonical Structure can_norm8 := MNorm norm8_pos norm8_hom ...
```

La distance sur les matrices est définie à partir de cette norme :

```
Variable R : numDomainType.
Let N8 := can_norm8 R.

Definition dist_mat m n (A B : 'M[R]_(m,n)) :=
 ||(A - B) : N8 |.
```

4.2.3 Preuve de l'axiome

Nous allons montrer ici le théorème que nous avons posé en axiome dans la première partie de la preuve du théorème de Perron-Frobenius :

Théorème 6. *Soit $A \in \mathcal{M}_n(\mathbb{K})$ ($\mathbb{K}$ étant $\mathbb{R}$ ou $\mathbb{C}$). La suite définit par $(A^k)_{k\in\mathbb{N}}$ converge vers 0 si et seulement si $\rho(A) < 1$.*

Nous verrons également quelques points de la formalisation de cette preuve.

La Preuve

Commençons d'abord par montrer le sens facile : Si la suite $(A^k)_{k\in\mathbb{N}}$ converge vers 0 alors $\rho(A) < 1$. Soit λ une valeur propre de la matrice A et x un vecteur propre associé à λ. Comme $Ax = \lambda x$ alors $A^k x = \lambda^k x$. Supposons que $(A^k)_{k\in\mathbb{N}}$ converge vers 0, alors la suite $(\lambda^k x)_{k\in\mathbb{N}} = (A^k x)_{k\in\mathbb{N}}$ converge vers 0 et comme x est non nul cela signifie que la suite $(\lambda^k)_{k\in\mathbb{N}}$ converge vers 0 et donc que $|\lambda| < 1$. Nous avons donc montré que pour toute valeur propre λ de la matrice A nous avons $|\lambda| < 1$ et donc en particulier nous avons $\rho(A) < 1$.

Maintenant supposons que $\rho(A) < 1$ et montrons que la suite $(A^k)_{k\in\mathbb{N}}$ converge vers 0. Notons $\mathcal{J}(A)$ la forme normale de Jordan de la matrice A. D'après ce que

[1] Le "8" représente un infini debout.

nous avons vu dans le chapitre 2, la matrice A est semblable à $\mathcal{J}(A)$. C'est-à-dire qu'il existe une matrice inversible P telle que $A = P^{-1}\mathcal{J}(A)P$, autrement dit telle que $A^k = P^{-1}\mathcal{J}(A)^k P$. Donc prouver que la suite $(A^k)_{k\in\mathbb{N}}$ converge vers 0 revient à prouver que la suite $(\mathcal{J}(A)^k)_{k\in\mathbb{N}}$ converge vers 0.

Nous savons que la forme normale de Jordan de A est une matrice diagonale par blocs dont chaque bloc est un bloc de Jordan $J(\lambda, n)$ où λ est une valeur propre de A. Donc la matrice $\mathcal{J}(A)^k$ est une matrice diagonale par blocs dont chaque bloc est de la forme $J(\lambda, n)^k$.

Ainsi pour montrer que la suite $(\mathcal{J}(A)^k)_{k\in\mathbb{N}}$ converge vers 0, il suffit de montrer que pour chaque bloc la suite des $(J(\lambda, n)^k)_{k\in\mathbb{N}}$ converge vers 0.

Voyons donc quelle est la forme générale de la matrice $J(\lambda, n)^k$. Sur la figure ci-dessous on peut voir ce qui se passe sur un exemple pour les quatre premières puissances avec $n = 4$:

$$J(\lambda,4) = \begin{pmatrix} \lambda & 1 & 0 & 0 \\ 0 & \lambda & 1 & 0 \\ 0 & 0 & \lambda & 1 \\ 0 & 0 & 0 & \lambda \end{pmatrix} \quad J(\lambda,4)^2 = \begin{pmatrix} \lambda^2 & 2\lambda & 1 & 0 \\ 0 & \lambda^2 & 2\lambda & 1 \\ 0 & 0 & \lambda^2 & 2\lambda \\ 0 & 0 & 0 & \lambda^2 \end{pmatrix}$$

$$J(\lambda,4)^3 = \begin{pmatrix} \lambda^3 & 3\lambda^2 & 3\lambda & 1 \\ 0 & \lambda^3 & 3\lambda^2 & 3\lambda \\ 0 & 0 & \lambda^3 & 3\lambda^2 \\ 0 & 0 & 0 & \lambda^3 \end{pmatrix} \quad J(\lambda,4)^4 = \begin{pmatrix} \lambda^4 & 4\lambda^3 & 6\lambda^2 & 4\lambda \\ 0 & \lambda^4 & 4\lambda^3 & 6\lambda^2 \\ 0 & 0 & \lambda^4 & 4\lambda^3 \\ 0 & 0 & 0 & \lambda^4 \end{pmatrix}$$

On remarque que sur chaque ligne les puissances de λ décroissent et que des coefficients binomiaux apparaissent. De manière générale nous avons :

$$(J(\lambda,n)^k)_{ij} = \binom{k}{j-i}\lambda^{k-(j-i)}$$

avec la convention que $\binom{k}{j-i} = 0$ si $j - i > k$ ou si $i > j$.

Rappelons que nous avons $\rho(A) < 1$ et donc pour toute valeur propre λ de A nous avons $|\lambda| < 1$. Nous allons montrer que si $|\lambda| < 1$ alors la suite $(J(\lambda, n)^k)_{k\in\mathbb{N}}$ converge vers 0, pour cela nous allons montrer que les coefficients de la matrice $J(\lambda, n)^k$ convergent vers 0 quand k tend vers l'infini.

Nous avons pour tout l :

$$\binom{k}{l} = \frac{k!}{(k-l)!\,l!} = \frac{(k-(l-1)) * \cdots * k}{l!} \leq \frac{k * \cdots * k}{l!} = \frac{k^l}{l!}$$

avec comme convention que $a - b = 0$ si $a < b$.

Cela signifie que pour un l fixé, $\binom{k}{l}$ est borné par le polynôme $k^l/l!$ et donc dans l'expression suivante :

$$\binom{k}{j-i}\lambda^{k-(j-i)}$$

comme $|\lambda| < 1$ nous avons le terme $\lambda^{k-(j-i)}$ qui converge vers 0 de façon exponentielle alors que le terme $\binom{k}{j-i}$ croit vers l'infini de façon polynômiale. Donc finalement chaque coefficient du bloc $J(\lambda, n)^k$ converge vers 0. Ceci fini de montrer le théorème.

Quelques points de formalisation

Matrice et convergence Nous allons d'abord parler de lemmes généraux sur la convergence des suites de matrices. Un premier lemme important que nous avons utilisé est celui qui fait le lien entre la convergence des matrices et la convergence coefficient par coefficient. Ce lemme s'énonce formellement ainsi :

```
Variable R : numFieldType.

Lemma mx_cvgP m n (U : nat -> 'M_(m,n)) (A :'M[R]_(m,n)) :
  U >->> A <-> (forall i j, (fun n => (U n) i j) >->> (A i j)).
```

Nous nous en sommes servi pour montrer que la suite des puissances des blocs de Jordan convergeait vers 0.

Le second résultat sur la convergence des matrices que nous allons voir est celui qui fait le lien entre la convergence des matrices diagonales par blocs et la convergence bloc à bloc. Rappelons qu'une matrice diagonale par blocs est définie formellement à l'aide d'une fonction `F : forall n, nat -> 'M_n.+1` qui donne les blocs diagonaux (voir 2.1), et d'une séquence qui correspond à peu de choses près à la taille des blocs. Une suite $U_1, U_2, \ldots$ de matrices diagonales par blocs correspondra formellement à une suite :

```
diag_block_mx s (UF 1), diag_block_mx s (UF 2), ...
```

où `UF` est la suite des fonctions qui donnent les blocs diagonaux. Nous avons donc `UF` qui est de type `nat -> forall n, nat -> 'M_n.+1`. Le lemme s'énonce ainsi :

```
Lemma diag_block_mx_cvg s (UF : nat -> forall n, nat -> 'M[R]_n.+1)
 (F : forall n, nat -> 'M[R]_n.+1) :
 (forall i, (i < size s)%N ->
   (fun k => UF k (nth 0%N s i) i) >->> (F (nth 0%N s i) i)) ->
 (fun k => diag_block_mx s (UF k)) >->> (diag_block_mx s F).
```

Nous l'utilisons pour montrer que la forme normale de Jordan de la matrice A converge vers 0.

Convergence sur les nombres réels ou complexes Nous avons également utilisé le fait qu'il y a équivalence entre $|\lambda| < 1$ et la convergence de la suite $(\lambda^k)_{k\in\mathbb{N}}$ vers zéro. La preuve de ce résultat utilise la propriété archimédienne des nombres réels. Or dans notre cas nous utilisons ces lemmes sur les nombres complexes qui n'est pas archimédien.

La structure `archifieldType` de la bibliothèque SSReflect est une structure `realFieldType` qui vérifie l'axiome suivant :

```
Definition archimedean_axiom (R : numDomainType) :=
    forall (x : R), exists ub : nat, `|x| < ub%:R.
```

La preuve de ce résultat n'utilise pas le fait que la structure de base est de type `realFieldType`, mais seulement le fait que l'axiome ci-dessus est vérifié. Or il se trouve que l'ensemble des nombres complexes vérifie cet axiome (car le module d'un nombre complexe est réel).

Pour éviter d'avoir à dupliquer les lemmes sur les nombres réels et sur les nombres complexes, nous allons définir une nouvelle structure plus faible que la structure `archiFieldType` dont le type de base sera la structure `numFieldType` :

```
Structure class_of (R : Type) := Class
  { base : NumField.class_of R;
        _ : archimedean_axiom (@NumDomain.Pack R base) }.

Coercion base : class_of >-> NumField.class_of.

Structure type := Pack {sort; _ : class_of sort}.
Coercion sort : type >-> Sortclass.
Notation archiDomainType := type.
```

Nous avons déjà rencontré ce genre de définition dans le chapitre 3 sur la présentation des structures topologiques. La structure `class_of` est la structure d'hérédité. Elle contient un champ `base` qui correspond à la structure d'hérédité du type de base, et un preuve que ce type de base vérifie les propriétés supplémentaires qui se trouve dans la structure `mixin_of` qui ici est seulement réduite à l'axiome `archimedean_axiom`. Ce genre de structure est expliquée plus en détail dans [15, 14].

Cette structure étant basée sur la structure `numFieldType` hérite des propriétés de cette dernière, et aussi de la topologie (et de la métrique) que nous avons instanciée pour la structure `numDomainType`.

L'ensemble des nombres réels et l'ensemble des nombres complexes étant tous les deux des instances de la structure ci-dessus, nous pouvons utiliser dans les deux cas les deux lemmes suivants qui nous intéressent :

```
Variable R : archiDomainType.

Lemma cvg_lt1 (x : R) :
  reflect ((fun k => x ^+ k) @->>> 0) (`|x| < 1).
Lemma polyM_exp_cvg0 i (a : R) :
 `|a| < 1 -> (fun k => k%:R ^+ i * a ^+ k) >->> 0.
```

4.3 Matrices à coefficients positifs ou nuls

Nous avons montré que pour toute matrice $A \in \mathcal{M}_n(\mathbb{R})$ telle que $0 < A$, il existe un vecteur propre x de A tel que $0 < x$ et que $Ax = \rho(A)x$. Nous allons voir

maintenant comment l'on généralise ce résultat pour le cas où $0 \leq A$. Ce que nous allons voir dans cette section n'est pas formalisé. Nous discuterons ici du travail qu'il reste à faire.

4.3.1 La preuve

Le théorème que nous allons étudier maintenant est donc le suivant :

Perron-Frobenius Soit $A \in \mathcal{M}_n(\mathbb{R})$ telle que $0 \leq A$, alors $\rho(A)$ est une valeur propre de la matrice A et possède un vecteur propre associé dont tous les coefficients sont positifs.

Prenons donc une matrice A telle que $0 \leq A$. Posons $A_k = (a_{ij} + \frac{1}{k})_{1 \leq i,j \leq n}$. Les matrices A_k sont des matrices strictement positives. Donc d'après le théorème que nous venons de montrer dans la section 4.2.1, nous avons $\rho(A_k)$ qui est une valeur propre de A_k et qui possède un vecteur propre associé x_k strictement positif. Quitte à normaliser, nous pouvons choisir le vecteur x_k de norme 1. Comme la suite des matrices A_k converge vers A, alors la suite des $\rho(A_k)$ converge vers $\rho(A)$. De plus, comme les vecteurs de la suite x_k sont dans un ensemble compact, alors d'après le théorème de Bolzano-Weierstraß (chapitre 3) il existe une sous-suite $x_{\varphi(k)}$ qui converge vers un vecteur x positif. Pour tout entier k nous avons donc $A_{\varphi(k)} x_{\varphi(k)} = \rho(A_{\varphi(k)}) x_{\varphi(k)}$, donc d'après ce qui précède, en passant à la limite nous obtenons $Ax = \rho(A)x$. Ainsi $\rho(A)$ est une valeur propre de A et possède un vecteur propre associé à coefficients positifs ou nuls.

Les deux principaux résultats de cette preuve qui demanderont le plus de travail de formalisation sont :

- Si $(A_n)_{n \in \mathbb{N}}$ est une suite de matrice qui converge vers A alors la suite des $\rho(A_n)$ converge vers $\rho(A)$.
- L'ensemble des vecteurs de norme 1 est un ensemble compact.

4.3.2 Convergence des rayons spectraux

Nous avons deux façons de montrer que la suite $(\rho(A_k))_{k \in \mathbb{N}^*}$ décrite plus haut converge vers $\rho(A)$. La première utilise un argument de convergence des suites réelles décroissantes et minorées, et la deuxième utilise un argument de continuité.

Argument de convergence

Pour montrer que la suite des rayons spectraux est décroissante, nous utilisons le lemme suivant :

Lemme 7. *Si $A, B \in \mathcal{M}_n(\mathbb{K})$ (avec $\mathbb{K} = \mathbb{R}$ ou $\mathbb{C}$) sont des matrices telles que $|A| \leq B$ alors $\rho(A) \leq \rho(B)$.*

Démonstration. Soit $\varepsilon > 0$. Définissons les matrices suivantes :

$$A_\varepsilon = \frac{A}{\rho(B) + \varepsilon} \text{ et } B_\varepsilon = \frac{B}{\rho(B) + \varepsilon}$$

Comme $|A| \leq B$, nous avons $|A_\varepsilon| \leq B_\varepsilon$ et donc $|A_\varepsilon|^k \leq B_\varepsilon^k$.

D'une part $\rho(B_\varepsilon) = \rho(B)/(\rho(B) + \varepsilon) < 1$. D'après ce que nous avons vu dans la section 4.2.3, cela signifie que la suite $(B_\varepsilon^k)_{k\in\mathbb{N}}$ converge vers 0.

D'autre part pour tout entier k nous avons $|A_\varepsilon^k| \leq |A_\varepsilon|^k \leq B_\varepsilon^k$. D'après ce qui précède, cela signifie que la suite $(A_\varepsilon^k)_{k\in\mathbb{N}}$ converge aussi vers 0, et donc cela implique que $\rho(A_\varepsilon) < 1$.

Donc finalement :

$$\rho(A_\varepsilon) = \frac{\rho(A)}{\rho(B) + \varepsilon} < 1 \Leftrightarrow \rho(A) < \rho(B) + \varepsilon$$

et ce pour tout $\varepsilon > 0$ donc $\rho(A) \leq \rho(B)$. □

Grâce à ce lemme nous pouvons montrer que la suite $(\rho(A_k))_{k\in\mathbb{N}^*}$ est une suite décroissante et minorée par $\rho(A)$ donc cette suite converge vers un réel ρ tel que $\rho \geq \rho(A)$.

De plus nous avons vu que en passant à la limite, nous avons l'égalité $Ax = \rho x$. Cela nous donne que $\rho \leq \rho(A)$ et donc par double inégalité $\rho = \rho(A)$.

Le développement sur la topologie est assez développé pour faire ce genre de preuve.

Argument de continuité

Une autre façon de faire est de montrer que la fonction $A \to \rho(A)$ est continue. Ainsi comme la suite $(A_k)_{k\in\mathbb{N}}$ converge vers A alors la suite $(\rho(A_k))_{k\in\mathbb{N}^*}$ converge vers $\rho(A)$.

L'argument le plus souvent utilisé pour montrer la continuité du rayon spectral est le suivant :

- Les coefficients du polynôme caractéristique d'une matrice varient continuement en fonction des coefficients de la matrice.
- Les racines d'un polynôme varient continuement en fonction des coefficients du polynôme.
- Le rayon spectral est continu en tant que composée d'applications continues.

Le polynôme caractéristique d'une matrice est le déterminant de sa matrice caractéristique. Donc une manière de prouver le premier point est d'instancier une métrique sur l'ensemble des polynômes et de prouver que l'application déterminant est continue.

On peut trouver une preuve du second point dans [34]. C'est une preuve qui demanderait encore du travail de formalisation, mais qui je pense reste abordable avec les outils que nous avons déjà formalisés.

Le troisème point est un résultat général sur la continuité qui est déjà formalisé dans le développement sur la topologie.

4.3.3 Compacité de la boule unité

Pour pouvoir utiliser le théorème de Bolzano-Weierstraß que nous avons vu au chapitre 3, il nous faut montrer que l'ensemble des vecteurs de norme 1 est un ensemble compact. Pour montrer ce résultat, on montre d'abord que les intervalles fermés sur $\mathbb{R}$ sont compacts.

Ensuite on montre que la boule unité est incluse dans un pavé qui est le produit cartésien d'intervalles de $\mathbb{R}$. Comme le produit cartésien fini d'ensembles compacts est compact, nous avons la boule unité qui est un ensemble fermé inclus dans un compact et est donc compact.

Une première étape serait de définir le produit cartésien de deux ensembles. Nous avons vu qu'il existe déjà dans Coq un type produit pour le produit cartésien de deux types (voir 1.1.1). Nous pourrions réutiliser ce produit cartésien sur les types pour avoir une définition du produit cartésien sur les ensembles :

```
Variables (T1 T2 : Type).
Definition cartesian_product (A : T1 -> Prop) (B : T2 -> Prop) :=
  fun x : (T1 * T2) => (x.1 'in A) /\ (x.2 'in B).
```

Mais je n'ai aucune idée si cette définition est praticable.

Un second problème va apparaître lorsque nous voudrons utiliser ce résultat sur l'ensemble des nombres complexes. Ici l'argument est que l'on peut voir l'ensemble $\mathbb{C}$ comme le produit cartésien $\mathbb{R} \times \mathbb{R}$. Il faudrait alors expliquer que la topologie obtenue à partir de la topologie produit sur $\mathbb{R} \times \mathbb{R}$ est équivalente à la topologie que nous avons instanciée sur l'ensemble des nombres complexes. Pour l'instant, telle qu'est définie la structure d'espace topologique, c'est quelque chose de difficile à exprimer.

Le fait qu'un ensemble fermé inclus dans un compact est compact est déjà formalisé dans le développement sur la topologie générale.

4.4 Conclusion

Le théorème de Perron-Frobenius est un exemple de théorème d'algèbre linéaire qui fait intervenir des arguments d'analyse et de topologie. Bien que la formalisation de ce théorème ait motivé la formalisation de la forme normale de Jordan et des espaces topologiques et métriques, ces développements ont été fait de manière indépendantes.

La premère partie du théorème a permis de montrer que ces développements étaient réutilisables. Nous avons pu ainsi avoir une formalisation complète du

théorème de Perron-Frobeniuspour les matrices strictements positives.

Toutefois pour montrer le théorème dans le cas générale, il reste encore à prouver des résultats comme la continuité du rayon spectrale ou la compacité de l'ensemble des vecteurs de norme 1.

4.5 Travaux reliés

A ma connaissance, il n'existe pas de travaux de formalisation à propos du théorème du théorème de Perron-Frobenius. Un des rares papiers que j'ai trouvé sur la formalisation de sujet semblable est celui de Karol Pąk où le concept de valeur propre est formalisé en Mizar [40].

Conclusion

La formalisation du théorème de Perron-Frobenius a permis de formaliser des résultats plus généraux d'algèbre linéaire, mais aussi de topologie générale. C'est ce théorème qui a principalement motivé cette thèse.

La bibliothèque SSREFLECT contient déjà une formalisation assez avancée sur les matrices, les polynômes et les propriétés algébriques. Ceci nous a fourni une bonne base pour développer les travaux de cette thèse.

La formalisation des formes normales de Frobenius et de Jordan vue au chapitre 2, outre le fait d'avoir une preuve formelle de ces résultats, a occasionné le développement de théories sur les matrices diagonales par blocs, et sur les matrices semblables ou équivalentes. Ce sont des théories qui sont utilisées dans d'autres contextes mathématiques et donc nous espérons que ces outils pourront être réutilisés pour la formalisation d'autres résultats mathématiques.

Dans le chapitre 3 nous avons formalisé les structures d'espace métrique et topologique. Il existe déjà une topologie dans la bibliothèque standard de COQ sur les nombre réels. Ici nous avons défini une structure générale de topologie pour avoir des résultats qui puissent être utilisés aussi bien sur des réels que sur des matrices ou sur des nombres complexes. Le résultat important de ce développement sur la topologie est le théorème de Bolzano-Weierstraß. Pour l'utilisation de ce théorème, il serait intéressant par la suite d'implémenter les structures d'espace normé et euclidien, ainsi que les propriétés de compacité des produits cartésiens d'ensembles. Cela permettrait d'avoir des théorèmes généraux supplémentaires pour montrer la compacité d'un ensemble.

Les développements sur les formes normales de matrices et sur la topologie ont été formalisés de manière indépendante et dans le but de pouvoir être réutilisés. Le théorème de Perron-Frobenius est une première application de l'utilisation de ces développements formels. En effet, nous avons réussi à instancier une métrique et une topologie sur les matrices et les nombres réels ou complexes. Ensuite nous avons pu prouver des résultats sur la convergence de certaines suites. Et la forme normale de Jordan a permis de montrer un résultat important sur la convergence des suites des puissances de matrices.

Cependant la partie du théorème de Perron-Frobenius utilisant le théorème de Bolzano-Weierstraß n'a pu être encore formalisée. Pour finir cette partie de la preuve, il reste à montrer quelques résultats comme la continuité du rayon spectral, et d'autres résultats de topologie pour pouvoir montrer la compacité d'un ensemble, et utiliser Bolzano-Weierstraß.

Bibliographie

[1] Andrea ASPERTI, Claudio Sacerdoti COEN, Ferruccio GUIDI, Enrico TASSI et Stefano ZACCHIROLI : Matita v0.99.1 : User manual, 2006. http://matita.cs.unibo.it/. 3

[2] Andrea ASPERTI, Maria Emilia MAIETTI, Claudio Sacerdoti COEN, Giovanni SAMBIN et Silvio VALENTINI : Formalization of formal topology by means of the interactive theorem prover matita. *In Proceedings of the 18th Calculemus and 10th International Conference on Intelligent Computer Mathematics*, MKM'11, pages 278–280, Berlin, Heidelberg, 2011. Springer-Verlag. 67

[3] Yves BERTOT et Pierre CASTÉRAN : *Interactive Theorem Proving and Program Development – Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. 9

[4] Yves BERTOT, Georges GONTHIER, Sidi Ould BIHA et Ioana PAŞCA : Canonical big operators. *In Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada*, volume 5170 de *Lecture Notes in Computer Science*, pages 86–101. Springer, 2008. 18

[5] François BOBOT, Sylvain CONCHON, Évelyne CONTEJEAN, Mohamed IGUERNELALA, Stéphane LESCUYER et Alain MEBSOUT : The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/. 3

[6] Robert Stephen BOYER et J Strother MOORE : *A Computational Logic Handbook*. Academic Press, Inc, 1988. 3

[7] Guillaume CANO et Maxime DÉNÈS : Matrices à blocs et en forme canonique. *In JFLA - Journées francophones des langages applicatifs*, Aussois, France, 2013. 25

[8] Cyril COHEN : *Formalized algebraic numbers : construction and first-order theory*. Thèse de doctorat, École Polytechnique, 2012. 23

[9] Cyril COHEN, Maxime DÉNÈS, Anders MÖRTBERG et Vincent SILES : Smith Normal form and executable rank for matrices, 2012.

http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples. 33, 35

[10] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki et Scott F. Smith : Implementing mathematics with the nuprl proof development system, 1986. 3

[11] Leonardo de Moura et Nikolaj Bjørner : Z3 : An efficient SMT solver. *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, 2008. http://z3.codeplex.com/. 3

[12] Maxime Dénès : *Étude formelle d'algorithmes efficaces en algèbre linéaire*. Thèse de doctorat, Université de Nice - Sophia Antipolis, 2013. 33

[13] The Coq development team : The Coq proof assistant : Reference manual, 2013. http://coq.inria.fr/. 3

[14] François Garillot : *Outils génériques de preuve et théorie des groupes finis*. These, Ecole Polytechnique X, décembre 2011. 59, 80

[15] François Garillot, Georges Gonthier, Assia Mahboubi et Laurence Rideau : Packaging Mathematical Structures. *In* Tobias Nipkow et Christian Urban, éditeurs : *Theorem Proving in Higher Order Logics*, volume 5674 de *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer. 80

[16] Isabelle Gil : *Contribution à l'algèbre linéaire formelle*. Thèse de doctorat, Institut Nationale Polytechnique de Grenoble, 1993. 25

[17] Ruben Gomboa, John Cowles et Jeff Van Baalen : Using ACL2 arrays to formalize matrix algebra. ACL2 Workshop 2003, Boulder, 2003. 50

[18] Georges Gonthier : The four colour theorem : Engineering of a formal proof. *In Computer Mathematics, 8th Asian Symposium, ASCM 2007*, volume 5081 de *Lecture Notes in Computer Science*, page 333. Springer, 2007. 4

[19] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Paşca, Laurence Rideau, Alexey Solovyev, Enrico Tassi et Laurent Théry : A machine-checked proof of the odd order theorem. *In Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France. Proceedings*, volume 7998 de *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013. 5

[20] Georges Gonthier et Assia Mahboubi : A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008. 4

[21] Thomas Callister Hales : Introduction to the flyspeck project. *In Mathematics, Algorithms, Proofs*, volume 05021 de *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. 5

[22] Thomas Callister Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua et Roland Zumkeller : A revision of the proof of the kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010. 5

[23] John Harrison : A HOL theory of Euclidean space. *In* Joe Hurd et Tom Melham, éditeurs : *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 de *Lecture Notes in Computer Science*, Oxford, UK, 2005. Springer-Verlag. 50, 67

[24] John Harrison : *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA, 1st édition, 2009. 3

[25] John Harrison : The HOL-Light system reference. University of Cambridge, 2013. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`. 3

[26] Johannes Hölzl, Fabian Immler et Brian Huffman : Type classes and filters for mathematical analysis in Isabelle/HOL. *In Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP'13, pages 279–294, Berlin, Heidelberg, 2013. Springer-Verlag. 67

[27] Katarzyna Jankowska : Matrices. Abelian group of matrices. *Formalized Mathematics*, 2(**4**):475–480, 1991. 50

[28] Matt Kaufmann et J Strother Moore : ACL2, 2013. `http://www.cs.utexas.edu/~moore/acl2/`. 3

[29] Matt Kaufmann, J. Strother Moore et Panagiotis Manolios : *Computer-Aided Reasoning : An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. 3

[30] David R. Lester : Topology in PVS : Continuous mathematics with applications. *In Proceedings of the Second Workshop on Automated Formal Methods*, AFM '07, pages 11–20, New York, NY, USA, 2007. ACM. 67

[31] Guillaume Melquiond : User's guide for gappa, 2013. `http://gappa.gforge.inria.fr/`. 3

[32] Jean-Pierre Merlet : Interval Analysis and Reliability in Robotics, mars 2006. 6, 69

[33] Michał Muzalewski : An outline of PC Mizar. Fondation Philippe le Hodey, 1993. http://mizar.uwb.edu.pl/. 3

[34] Raúl Naulin et Carlos Pabst : The roots of a polynomial depend continuously on its coefficients. *Revista Colombiana de Matemáticas*, 28(1):35–37, 1994. 83

[35] Michael Norrish, Konrad Slind *et al.* : The HOL system reference. Automated Reasoning Group, University of Cambridge, Computer Laboratory, 2013. http://hol.sourceforge.net/. 3

[36] Steven Obua : Proving bounds for real linear programs in Isabelle/HOL. *In* Joe Hurd et Thomas F. Melham, éditeurs : *TPHOLs*, volume 3603 de *Lecture Notes in Computer Science*, pages 227–244. Springer, 2005. 50

[37] Patrick Ozello : *Calcul exact des formes de Jordan et de Frobenius d'une matrice*. Thèse de doctorat, Université Scientifique Technologique et Médicale de Grenoble, 1987. 25

[38] Beata Padlewska et Agata Darmochwał : Topological spaces and continuous functions. *Formalized Mathematics*, 1(**1**):223–230, 1990. 67

[39] Karol Pąk : Block diagonal matrices. *Formalized Mathematics*, 16(**3**):259–267, 2008. 50

[40] Karol Pąk : Eigenvalues of a linear transformation. *Formalized Mathematics*, 16(**4**):289–295, 2008. 84

[41] Karol Pąk : Jordan matrix decomposition. *Formalized Mathematics*, 16(**4**):297–303, 2008. 50

[42] Ioana Paşca : *Vérification formelle pour les méthodes numériques*. Thèse de doctorat, Université Nice-Sophia Antipolis, 2010. 6, 69, 73, 77

[43] Giovanni Sambin : The basic picture, a structure for topology (the basic picture, i). Rapport technique, 2001. 67

[44] Stephan Schulz : E 1.8 : User manual, 2013. http://www4.informatik.tu-muenchen.de/~schulz/E/. 3

[45] Natarajan Shankar, Sam Owre, John M. Rushby et David William John Stringer-Calvert : PVS prover guide, 2001. http://pvs.csl.sri.com/. 3

[46] Vincent Siles : A formal proof of the Cauchy-Binet formula, 2012. http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples. 38

[47] Matthieu SOZEAU et Nicolas OURY : First-Class Type Classes. *In* Otmame AÏT-MOHAMED, César MUÑOZ et Sofiène TAHAR, éditeurs : *21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 de *Lecture Notes in Computer Science*. Springer, août 2008. 53

[48] Shlomo STERNBERG : *Dynamical Systems*. Dover Publications, 2010. Chapitre 9. 69

[49] Joe Hendrix UNIVERSITY et Joe HENDRIX : Matrices in ACL2, 2003. 50

[50] Makarius WENZEL *et al.* : The Isabelle/Isar reference manual, 2013. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`. 3