Caractérisation de PSPACE par équations différentielles

Yassine Hamoudi
encadré par Olivier Bournez

2 juin - 11 juillet, 2014

Résumé

Ce rapport de stage présente un résultat de complexité portant sur les modèles de calcul à temps et espace continus. Il y est exposé plusieurs tentatives de caractérisation de la classe PSPACE à partir des équations différentielles à second membre polynomial. L'objectif poursuivi par ce travail est l'enrichissement des connaissances acquises en calculabilité et en complexité concernant le modèle GPAC (General Purpose Analog Computer).

L'ensemble des théorèmes issus de travaux antérieurs sont précédés de leurs références bibliographiques. Ceci permet de repérer facilement les résultats produits effectivement au cours du stage.

Table des matières

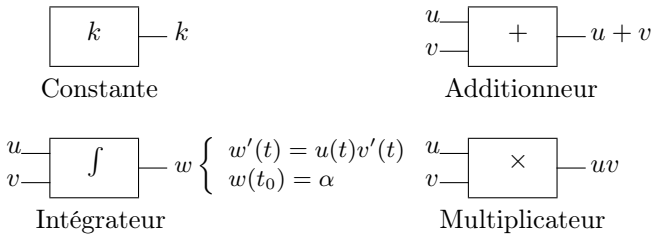


FIGURE 1 – Briques de base du GPAC

1 Introduction

Les modèles de calcul à temps et espace discrets occupent une place centrale en théorie de la complexité et de la calculabilité. Leur grande diversité (machines de Turing, λ-calcul, fonctions récursives...) et leur unification à travers la thèse de Church-Turing participent à leur popularité. Il existe également une théorie des modèles continus, qui inclue par l'exemple l'analyse récursive, les machines de Blum-Shub-Smale ou le modèle GPAC. Qu'advient-il alors lorsque temps et espace deviennent continus ? Peut-on établir des résultats similaires à ceux connus dans le cadre discret, voir rapprocher ces modèles de la thèse de Church-Turing ? Les résultats dans ce domaine sont aujourd'hui peu nombreux et il reste à développer une théorie aussi riche et élégante que dans le cadre discret.

Notre étude va porter sur un modèle analogique particulier, appelé *modèle GPAC (General Purpose Analog Computer)*. Développé à partir de 1941 par Shannon, il propose pour cadre de travail les équations différentielles ordinaires à second membre polynomial. Des liens ont déjà été établis entre ce modèle et l'analyse récursive dans [BCGH06], ou les machines de Turing dans [BGP14]. Des résultats en complexité, figurant dans [BGP14], ont permis récemment de caractériser les classes P et NP via GPAC. Il s'agit de prolonger ces études dans le champs de la complexité, en proposant une caractérisation de la classe PSPACE par GPAC.

Nous chercherons dans un premier temps à définir la classe PSPACE en terme de temps de calcul sur une machine discrète particulière. Nous serons emmenés pour cela à modifier le modèle RAM habituel. Il s'agira ensuite de simuler une telle machine par équations différentielles. Nous prolongerons à cette fin le travail effectué dans [BGP14], qui propose la simulation d'une machine de Turing par équations différentielles.

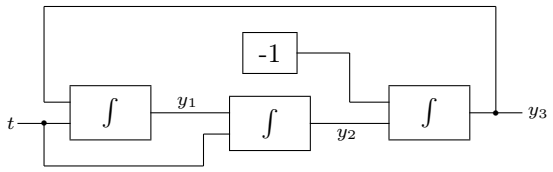
Le modèle GPAC et quelques uns des principaux résultats le concernant sont présentés partie 2, une première caractérisation de PSPACE par équations différentielles y est également exposée. Nous détaillons ensuite, partie 3, trois modèles de calcul discrets inspirés des machines RAM. Des liens sont établis entre PSPACE et ces derniers, en partie 4. Enfin, une caractérisation plus aboutie de la classe PSPACE est exposée partie 5, où nous essaierons de simuler par équations différentielles une machine discrète définie pour l'occasion : les automates arithmétiques.

2 Le modèle GPAC et les équations différentielles ordinaires

Le modèle GPAC est un modèle de calcul à temps et espace continus, défini à partir des équations différentielles ordinaires. Quelques définitions et résultats s'y rapportant sont présentés ci-dessous.

2.1 Présentation du GPAC

Le modèle GPAC (General Purpose Analog Computer) fut étudié pour la première fois par Claude Shannon en 1941. Il s'agit d'un modèle mathématique d'une machine analogique : l'analyseur différentiel. Dans sa définition originelle, le GPAC est présenté comme étant une famille de circuits obtenus à partir des quatre briques de base représentées figure 1. Un assemblage judicieux de ces briques permet de générer certains fonctions, telles que les fonctions sin et cos représentées figure 2.



$$\begin{cases} y_1' = y_3 & \& \quad y_1(0) = 1 \\ y_2' = y_1 & \& \quad y_2(0) = 0 \\ y_3' = -y_2' & \& \quad y_3(0) = 0 \end{cases}$$

$y_1 = \cos(t)$, $y_2 = \sin(t)$, $y_3 = -\sin(t)$.

FIGURE 2 – sin et cos sont calculables par GPAC

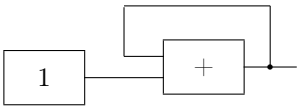


FIGURE 3 – Circuit n'admettant pas de sortie possible

Toutefois, cette définition est difficile à manier et peut donner lieu à des circuits inconsistants (cf figure 3). Nous considérerons donc par la suite une classe particulière de GPAC, présentée définition ci-dessous.

Définition 1. Une fonction $f : \mathbb{R} \to \mathbb{R}$ est *calculable par GPAC* si et seulement si elle est une composante de la solution à un système :

$$\begin{cases} y'(t) = p(y(t)) \\ y(0) = y_0 \end{cases} \tag{1}$$

où $p : \mathbb{R}^d \to \mathbb{R}^d$ est un vecteur de polynômes.

Il est possible d'associer à toute fonction calculable par GPAC un circuit calculant effectivement cette fonction (la figure 2 en témoigne pour les fonctions sin et cos), toutefois, la réciproque n'est pas vraie. On définit également la notion de langage reconnu.

Définition 2. Un langage $\mathcal{L}$ est reconnu par GPAC si et seulement si il existe un vecteur de polynômes p et une fonction d'encodage ψ tel que pour tout mot w, la solution y au système :

$$\begin{cases} y' = p(y) \\ y(0) = \psi(w) \end{cases}$$

satisfait :

— y est définie sur $\mathbb{R}$ `[existence]`
— si $y_1(t) \geqslant 1$ (resp. $y_1(t) \leqslant -1$) alors $\forall u \geqslant t$, $y_1(u) \geqslant 1$ (resp. $y_1(u) \leqslant -1$) `[stabilité]`
— si $w \in \mathcal{L}$ (resp. $w \notin \mathcal{L}$) alors $\exists t > 0$ t.q. $y_1(t) \geqslant 1$ (resp. $y_1(t) \leqslant -1$) `[accessibilité]`

La classe des langages reconnus par GPAC est appelée R_{GPAC}.

2.2 Résultats de calculabilité et de complexité sur les GPAC

La théorie des GPAC a été relativement peu étudiée jusqu'à présent. Cela laisse une grande liberté dans les recherches à entreprendre, mais nécessite également d'établir certains résultats fondamentaux en calculabilité et en complexité. Différents théorèmes permettent d'ors et déjà de caractériser les langages reconnus par GPAC, et de rapprocher ce modèle d'autres machines préexistantes.

Ainsi, il a été prouvé ([GCB08] et [CG09]) que la classe des langages reconnus par GPAC est exactement l'ensemble des langages récursifs. Des résultats en complexité ont également été établis récemment ([BGP14]). Nous les présentons ci-dessous.

Définition 3 (Longueur d'un arc)**.** Soit $I = [a, b]$ un intervalle fini et $y : I \to \mathbb{R}^n$ un arc rectifiable. On définit la *longueur* de l'arc sur I par :

$$\text{length}(y)(I) = \int_a^b \|y'(t)\|_2 dt$$

où

$$\|(x_1, \ldots, x_n)\|_2 = \sqrt{|x_1|^2 + \cdots + |x_n|^2}$$

La longueur d'arc est l'outil utilisé pour mesurer le temps de calcul sur un GPAC. Il permet notamment de caractériser le temps polynomial :

Définition 4. Un langage $\mathcal{L}$ est reconnu en temps polynomial par un GPAC si et seulement si il existe un vecteur de polynômes p, une fonction d'encodage ψ et un polynôme q tel que pour tout mot w, la solution y au système :

$$\begin{cases} y' = p(y) \\ y(0) = \psi(w) \end{cases}$$

satisfait :

— y est définie sur $\mathbb{R}$
— si $y_1(t) \geqslant 1$ (resp. $y_1(t) \leqslant -1$) alors $\forall u \geqslant t$, $y_1(u) \geqslant 1$ (resp. $y_1(u) \leqslant -1$)
— si $w \in \mathcal{L}$ (resp. $w \notin \mathcal{L}$) alors $y_1(t) \geqslant 1$ (resp. $y_1(t) \leqslant -1$) dès que $\text{length}(y)([0,t]) \geqslant q(|w|)$
— $\forall t \geqslant 0$, $\text{length}(y)([0,t]) \geqslant t$

La classe des langages reconnus par GPAC en temps polynomial est appelée P_{GPAC}.

Théorème 5 ([BGP14])**.** L'égalité suivante est vérifiée : $P_{GPAC} = P$

Les restrictions sur la longueur de la courbe ajoutées dans la définition 4 (par rapport à la définition 2) permettent d'obtenir l'inclusion $P_{GPAC} \subset P$.

La classe NP_{GPAC} a également été définie (voir [BGP14]) et son égalité avec NP est prouvée. Ainsi, le modèle GPAC vérifie la thèse de Church-Turing aussi bien au niveau de la calculabilité que de la complexité. En particulier, les résultats précédents donnent une formulation en terme d'équations différentielles ordinaires de la question $P = NP$?.

Nous nous attacherons par la suite à essayer de prolonger ces résultats en étudiant dans quelle mesure la classe PSPACE peut être caractérisée par GPAC.

2.3 Simulation d'une machine de Turing par équations différentielles

On présente de manière succincte le travail effectué dans [BGP14] afin de simuler une machine de Turing par équations différentielles. Ce résultat permet d'affirmer $P \subseteq P_{GPAC}$ et constitue la base des travaux que nous exposerons par la suite.

Considérons une machine de Turing à ruban semi-infini $M = (Q, \Sigma, \delta, q_0, F)$ où $Q = \{0, ..., m-1\}$ est l'ensemble des états, $F \subseteq Q$ est l'ensemble des états acceptants, $q_0 \in Q$ est l'état initial et $\Sigma = \{0, ..., k-2\}$ est l'alphabet. On définit la taille du ruban comme étant la plus petite position à partir de laquelle il n'y a plus que des symboles blancs. On a $\delta = (\delta_1, \delta_2, \delta_3, \delta_s) : Q \times \Sigma \to Q \times \Sigma \times \{L, R\} \times \{-1, 0, 1\}$ la fonction de transition avec $L = 0$ (déplacement de la tête de lecture à gauche) et $R = 1$ (déplacement à droite), tel que δ_1 indique le nouvel état, δ_2 le symbole à écrire avant déplacement, δ_3 le sens de déplacement et δ_s l'évolution de la taille du ruban.

On définit une configuration c comme étant un 5-uplet $c = (x, s, y, q, z)$ où x est la partie du ruban située à gauche de la tête de lecture, y la partie située à droite, s le symbole placé sous la tête de lecture, q l'état et z la taille du ruban. On travaille avec un encodage rationnel noté $[c] = (0.x, s, 0.y, q, z)$ où $0.x = x_0 k^{-1} + x_1 k^{-2} + ... + x_n k^{-n-1}$.

On souhaite définir une fonction $\text{step}_{\mathcal{M}} : \mathbb{R}^5 \to \mathbb{R}^5$ tel que pour toute configuration c, $\text{step}_{\mathcal{M}}^{[n]}([c])$ corresponde à la configuration de la machine après n pas de calcul ($\text{step}_{\mathcal{M}}^{[n]}$ désigne la fonction $\text{step}_{\mathcal{M}}$ itérée n fois). La manière la plus naturelle d'effectuer cela est d'utiliser des interpolations des différentes fonctions de transition. Pour une fonction $f : G \to \mathbb{R}$ où G est un sous-ensemble de $\mathbb{R}^d$ fini, on considère son interpolation de Lagrange L_f définie par :

$$L_f(x) = \sum_{\bar{x} \in G} f(\bar{x}) \prod_{i=1}^{d} \prod_{\substack{y \in G \\ y \neq \bar{x}}} \frac{x_i - y_i}{\bar{x}_i - y_i}$$

On a alors :

$$\text{step}_{\mathcal{M}} \begin{pmatrix} x \\ s \\ y \\ q \\ z \end{pmatrix} = \begin{pmatrix} \text{choose}\left[\text{frac}(kx), \frac{x+L_{\delta_2}(q,s)}{k}\right] \\ \text{choose}\left[\text{int}(kx), \text{int}(ky)\right] \\ \text{choose}\left[\frac{y+L_{\delta_2}(q,s)}{k}, \text{frac}(ky)\right] \\ L_{\delta_1}(q,s) \\ z + L_{\delta_s}(q,s) \end{pmatrix}$$

où

$$\text{choose}[a,b] = (1 - L_{\delta_3}(q,s)) \cdot a + L_{\delta_3}(q,s) \cdot b$$

int désigne la partie entière et frac la partie fractionnaire

Malheureusement, cette fonction ne peut pas être itérée en l'état par une équation différentielle ordinaire à second membre polynomial. Il faut tout d'abord utiliser des fonctions analytiques en remplacement, et en approximation, de int et frac. Par exemple, on peut approximer la partie entière par $\sigma_\infty(x) = x - \frac{1}{2\pi}\sin(2\pi x)$. Les erreurs introduites par cela doivent rester suffisamment faibles et permettre à la simulation de fonctionner.

Afin de contrôler les marges d'erreur, on utilise des machines de Turing Zig-Zag. Ces machines ont la particularité d'atteindre régulièrement un des bords du ruban. Lorsqu'un bord est atteint, on peut affirmer avec certitude que, dans la configuration courante, $x = 0$ (bord gauche atteint) ou $y = 0$ (bord droit atteint). Ceci permet de rétablir régulièrement des valeurs exactes pour x et y dans la simulation.

L'ensemble des techniques précédentes permettent de construire une fonction $\overline{\text{step}_{\mathcal{M}}} : \mathbb{R}^5 \to \mathbb{R}^5$ qui approxime suffisamment bien la fonction $\text{step}_{\mathcal{M}}$ recherchée, et qui soit peu sensible aux perturbations. Cette fonction est présentée plus en détail dans [BGP14], définition 55. L'équation différentielle permettant d'itérer $\text{step}_{\mathcal{M}}$ y est également exposée, lemme 64. Nous ne la reprenons pas ici, toutefois, nous en présenterons une version modifiée lemme 35. Ce dernier lemme expose notamment les conditions de stabilité que doit vérifier la fonction à itérer, et qui restreignent la construction de $\overline{\text{step}_{\mathcal{M}}}$.

Il est ainsi possible de simuler une machine de Turing par GPAC. On démontre alors facilement que tout langage reconnu en temps polynomial par machine de Turing l'est aussi par GPAC (au sens de la définition 4). Ceci permet de conclure que $P \subseteq P_{GPAC}$. L'autre inclusion s'obtient par la manipulation d'outils mathématiques spécifiques aux équations différentielles, que nous n'évoquerons pas dans ce rapport.

2.4 Première caractérisation de PSPACE par équations différentielles

Si on équipe une machine de Turing d'un oracle pouvant résoudre un problème PSPACE-complet en temps constant, alors cette machine reconnaît les langages de PSPACE en temps polynomial. Ceci nous permet de produire une première caractérisation de PSPACE par équations différentielles.

Définition 6. Une *machine de Turing à oracle* est un 9-uplet $M = (Q, \Sigma, \delta, q_0, F, q_{ask}, q_N, q_Y, O)$ où Q, Σ, δ, q_0, et F sont identiques à ce qui a été présenté partie précédente, $O : \Sigma^* \to \{q_N, q_Y\}$ est la fonction d'oracle qui associe à tout mot $w \in \Sigma^*$ l'état q_Y si le mot est accepté par l'oracle, q_N sinon. q_{ask} est un état d'appel à l'oracle.

On définit comme précédemment la configuration $c = (x, s, y, q, z)$. Le ruban d'une machine à oracle est constitué d'un mot destiné à être fourni à l'oracle, suivi du symbole #, puis de la partie de travail habituelle. # appartient à Σ mais n'est utilisé que pour séparer le mot d'oracle du ruban de travail.

$$\underbrace{w_0 w_1 w_2 ... w_n}_{\substack{\text{mot pouvant être fourni} \\ \text{à l'oracle à tout instant}}} \# \underbrace{s_0 s_1 s_2 ....}_{\substack{\text{zone de travail} \\ \text{habituelle}}}$$

Lorsque la machine entre dans l'état q_{ask} la partie du ruban située à gauche de # est fournie immédiatement à l'oracle O, dont la réponse détermine le nouvel état.

Afin de faciliter la simulation par GPAC, on suppose que l'appelle à l'oracle ne peut se faire que lorsque la tête de travail est placée sur # (ainsi le mot d'oracle est situé immédiatement à gauche de la tête). On peut alors appeler systématiquement l'oracle sur la partie située à gauche de la tête, mais ne retenir sa réponse que lorsque l'on est dans l'état q_{ask}. On suppose enfin que la fonction d'oracle O est prolongée sur $\mathbb{R}$, et que q_{ask} est encodé par 0.

La fonction $\text{step}_{\mathcal{M}}$ se modifie alors facilement :

$$\text{step}_{\mathcal{M},\mathcal{O}}\begin{pmatrix} x \\ s \\ y \\ q \\ z \end{pmatrix} = \begin{pmatrix} \text{choose}_{\text{O}}\left[\text{choose}\left[\text{frac}(kx), \frac{x+L_{\delta_2}(q,s)}{k}\right], x\right] \\ \text{choose}_{\text{O}}\left[\text{choose}\left[\text{int}(kx), \text{int}(ky)\right], s\right] \\ \text{choose}_{\text{O}}\left[\text{choose}\left[\frac{y+L_{\delta_2}(q,s)}{k}, \text{frac}(ky)\right], y\right] \\ \text{choose}_{\text{O}}\left[L_{\delta_1}(q,s), O(x)\right] \\ \text{choose}_{\text{O}}\left[z + L_{\delta_s}(q,s), z\right] \end{pmatrix}$$

où

$$\text{choose}_{\text{O}}[a,b] = (1 - \mathbb{1}(q)) \cdot a + \mathbb{1}(q) \cdot b$$

$$\mathbb{1} \text{ vaut 1 en 0 } (= q_{ask}) \text{ et 0 partout ailleurs}$$

On peut ensuite appliquer les techniques d'approximation présentées brièvement en 2.3 et dans [BGP14] (la surcouche introduite par l'oracle dans la fonction d'itération ne nécessite pas l'usage de fonctions rendant incontrôlables les marges d'erreur). Les équations différentielles permettant de simuler une machine de Turing avec oracle O sont alors de la forme $y'(t) = p(y, O(y))$.

Nous obtenons le théorème suivant :

Théorème 7. Si $\mathcal{L}$ est un langage reconnu en temps polynomial par une machine de Turing avec oracle O, alors il existe une fonction (d'oracle) $o : \mathbb{R}^d \to \mathbb{R}^d$ tel que $\mathcal{L}$ est reconnu en temps polynomial (au sens de la définition 4) par une équation différentielle de la forme :

$$y'(t) = p(y, o(y))$$

où $p : \mathbb{R}^{d^2} \to \mathbb{R}^d$ est un vecteur de polynômes.

Corollaire 8. PSPACE est inclue dans l'ensemble des langages reconnus en temps polynomial par une équation différentielle de la forme :

$$y'(t) = p(y, o(y))$$

où p est un vecteur de polynômes et o une fonction capable d'associer à tout encodage réel d'une *Quantified boolean formula* (QBF) la valeur 1 si la QBF est vraie, 0 sinon.

Démonstration. Le problème QBF est PSPACE complet. Une machine de Turing munie d'un oracle capable de répondre à ce problème peut donc résoudre tout problème de PSPACE en temps polynomial. D'après le théorème 7, ceci permet bien d'obtenir une caractérisation de PSPACE par équations différentielles. □

Cette première caractérisation n'est cependant pas satisfaisante en l'état. En effet, les fonctions d'oracles et les encodages utilisés ne sont pas explicités (Comment encoder une QBF dans $\mathbb{R}$? Comment expliciter une fonction d'oracle travaillant sur des QBF ?). L'enrichissement des équations différentielles par des fonctions d'oracle o nécessiterait plus d'informations, et de restrictions, sur ces fonctions pour être un choix pertinent. Nous allons donc adopter une toute autre démarche par la suite, afin d'obtenir une caractérisation plus élégante de PSPACE par équations différentielles.

3 Le modèle RAM et ses dérivés

Les modèles de calcul discrets sont nombreux à exister et comportent chacun leurs propres intérêts en théorie de la complexité et de la calculabilité. Nous présentons ici trois modèles de calcul proches des machines RAM (*Random Access Machines*). Ceux-ci ont la particularité de pouvoir caractériser PSPACE en terme de temps de calcul, comme nous le verrons partie suivante.

3.1 Instructions de base et machines RAM

Les programmes exécutables par machine RAM (et modèles dérivés) consistent en une suite finie d'instructions numérotées à partir de 0 et opérant sur des registres R_1, R_2, R_3.... L'ensemble des instructions permises dépend du modèle de calcul considéré.

Les *instructions de bases* représentées figure 4 sont communes à l'ensemble des machines que l'on utilisera.

Instruction	Description
$R_a \leftarrow x$	Chargement direct
$if\ R_a = 0\ goto\ j$	Conditionnelle (sauter directement à la $j^{ème}$ instruction si $R_a = 0$)
$if\ R_a \neq 0\ goto\ j$	Conditionnelle (sauter directement à la $j^{ème}$ instruction si $R_a \neq 0$)
$accept$	Acceptation (état d'arrêt)
$reject$	Rejet (état d'arrêt)

FIGURE 4 – Instructions de base

On présente le modèle RAM classique que l'on va modifier par la suite.

Définition 9. Une *machine RAM* est constituée d'une mémoire de registres R_1, R_2..., chacun pouvant contenir un entier de taille *bornée*, et d'un programme P consistant en une liste finie d'instructions numérotées à partir de 0. Les instructions autorisées sont constituées des instructions de base (fig. 4) enrichies de :

Instruction	Description
$R_a \leftarrow R_{R_b}$	Lecture indirecte
$R_{R_a} \leftarrow R_b$	Chargement indirect
$R_a \leftarrow R_a + 1$	Successeur

L'exécution d'un programme P sur un mot w consiste à placer w dans un registre d'entrée (R_1 par exemple), puis à exécuter P jusqu'à entrer dans un état d'arrêt.

3.2 Machines à Vecteurs

Les machines à Vecteurs sont un modèle de calcul parallèle. Contrairement aux machines RAM précédentes, elles peuvent stocker des entiers de taille non bornée et effectuer des opérations bit à bit.

Définition 10. Une *machine à Vecteurs* est constituée d'une mémoire de registres R_1, R_2..., chacun pouvant contenir un entier *relatif* de taille *arbitraire*, et d'un programme P consistant en une liste finie d'instructions numérotées à partir de 0. Les instructions autorisées sont constituées des instructions de base (fig. 4) enrichies de :

Instruction	Description
$R_a \leftarrow \neg R_a$	Négation bit à bit
$R_a \leftarrow R_b \wedge R_c$	Conjonction bit à bit
$R_a \leftarrow R_b \uparrow R_c$	Shift gauche
$R_a \leftarrow R_b \downarrow R_c$	Shift droite

Lorsque R_c est positif, l'opération `shift gauche` consiste à décaler R_b vers la gauche en ajoutant R_c fois 0 à la droite de R_b. Lorsque R_c est négatif, R_b est décalé vers la droite en supprimant ses R_c bits de poids les plus faibles. Le `shift droite` est l'opération inverse.

Les entiers positifs se terminent par une séquence de 0 (par exemple 5 est représenté par ...000101), les entiers négatifs s'obtiennent en complémentant leur opposé (par exemple -5 est représenté par ...111010). La taille d'un nombre est la taille de sa partie non constante.

Exemple 11. Quelques calculs élémentaires :
— $5 \wedge 12 = (...000101)_2 \wedge (...0001100)_2 = (...0001101)_2 = 13$
— $\neg 10 = \neg(...000101)_2 = (...111010)_2 = -5$
— $3 \uparrow 2 = (...00011)_2 \uparrow 2 = (...0001100)_2 = 12$
— $9 \downarrow 2 = (...0001001)_2 \downarrow 2 = (...00010)_2 = 2$

Exemple 12. Les opérations bit à bit et le shift permettent d'effectuer certains calculs très rapidement. Par exemple, le programme suivant (voir [BDG90]) prend en entrée un mot β de longueur s et produit le mot β^k où $k = 2^r$ en $O(r)$ opérations unitaires (`while` s'obtient à partir des instructions permises).

$$\texttt{V} \leftarrow \beta \texttt{ ; S} \leftarrow s \texttt{ ; L} \leftarrow 2^r \texttt{ ;}$$
$$\texttt{while } L > 1 \texttt{ do V} \leftarrow \texttt{V} \vee \texttt{(V} \uparrow \texttt{S) ; S} \leftarrow \texttt{S} \uparrow \texttt{1 ; L} \leftarrow \texttt{L} \downarrow \texttt{1 od}$$

Toutefois, il est possible avec ce modèle d'obtenir très rapidement des nombres particulièrement grand. Par exemple, le programme suivant construit en n étapes un nombre supérieur à $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (2 est répété n fois) : `V` $\leftarrow 1$ `; do` n `times V` $\leftarrow$ `V`$\uparrow$`V od`. Afin de restreindre la taille des vecteurs (et pouvoir obtenir des résultats intéressants par la suite), on modifie légèrement le modèle : on distingue *dorénavant* deux types de registres : les vecteurs, notés $V_1, V_2...$, et les indexes, notés $I_1, I_2....$ Les seules opérations de shift autorisées sont : $V_i \leftarrow V_j \uparrow I_k$, $V_i \leftarrow V_j \downarrow I_k$, $I_i \leftarrow I_j \uparrow 1$ et $I_i \leftarrow I_j \downarrow 1$. Par ailleurs, les opérations booléennes ne doivent s'effectuer qu'entre vecteurs ou qu'entre indexes. Enfin, tout mot donné en entrée à un programme doit être placé dans un vecteur.

Avec les modifications précédentes, il est aisé de constater qu'il existe deux constantes p et q (dépendantes du programme P exécuté, mais indépendantes de l'entrée) tel que la taille de toute index est bornée par $p + t$ au temps t, et la taille de tout vecteur par $2^{p+t} + q$.

3.3 Machines RAM arithmétiques

Les opérations bit à bit des machines à Vecteurs vont poser problème par la suite. Par conséquent, nous allons définir une alternative à ces machines en repartant à nouveau du modèle RAM, et en ajoutant cette fois-ci des opérations arithmétiques et des registres non bornés.

Définition 13. Une *machine RAM arithmétique* est constituée d'une mémoire de registres $R_1, R_2...$, chacun pouvant contenir un entier de taille *arbitraire*, et d'un programme P consistant en une liste finie d'instructions numérotées à partir de 0. Les instructions autorisées sont constituées des instructions de base (fig. 4) enrichies de :

Instruction	Description
$R_a \leftarrow R_{R_b}$	Lecture indirecte
$R_{R_a} \leftarrow R_b$	Chargement indirect
$R_a \leftarrow R_b + R_c$	Addition
$R_a \leftarrow R_b - R_c$	Soustraction ($a - b = max(0, a - b)$)
$R_a \leftarrow R_b \times R_c$	Multiplication
$R_a \leftarrow R_b \div R_c$	Division entière

Remarque 14. On notera mod l'opération : $x \bmod y = x - y \times (x \div y)$

3.4 Automates arithmétiques

Les opérations d'adressages indirectes des RAM arithmétiques ne permettent pas de borner le nombre de registres nécessaires à un programme. Afin de contourner ce problème, nous allons essayer de simuler les machines RAM arithmétiques en se restreignant à des opérations sur registres adjacents, comme cela se fait pour les automates cellulaires. L'objectif est d'associer à toute machine RAM arithmétique M une fonction δ_M particulière, et définir un pas de calcul comme étant le remplacement de chaque registre R_i par la valeur $\delta_M(R_{i-1}, R_i, R_{i+1})$. L'exécution d'un programme consistera alors en une suite de pas de calcul menant à un état d'arrêt.

Nous avons été emmenés pour cela à construire le modèle suivant :

Définition 15. Un *automate arithmétique* est un 2-uplet (Q, δ) où Q est un sous-ensemble fini de $\mathbb{N}$ et $\delta : (Q \times \mathbb{N})^3 \to Q \times \mathbb{N}$ est une *fonction locale d'évolution* vérifiant :

— il existe $\delta_Q : Q^3 \times \{0,1\}^3 \to Q$,

— il existe $\delta_{\mathbb{N},1}, ...\delta_{\mathbb{N},s} : Q^3 \times \{0,1\}^3 \to \{0,1\}$,

tels que :

$$\delta((q_0, v_0), (q_1, v_1), (q_2, v_2)) = (\delta_Q(\Gamma), \sum_{i=1}^{s} \delta_{\mathbb{N},i}(\Gamma) \cdot A_i)$$

où :

— $\Gamma = (q_0, q_1, q_2, \mathbb{1}(v_0), \mathbb{1}(v_1), \mathbb{1}(v_2))$ ($\mathbb{1}$ vaut 1 en 0 et 0 partout ailleurs)

— $A_i = x$ ou $A_i = x\ op\ y$ avec $x, y \in \{1, v_0, v_1, v_2\}$ et $op \in \{+, -, \times, \div\}$ (s représente le nombre total de combinaisons possibles pour A_i, soit 68)

Définition 16. On appelle *configuration* d'un automate arithmétique tout élément $c \in (Q \times \mathbb{N})^{\mathbb{Z}}$. On note $(q_i, v_i) = c_i$ le contenu de la position i de l'automate, q_i désigne l'état et v_i la valeur de cette position.

On appelle *fonction globale d'évolution* la fonction $F_\delta : (Q \times \mathbb{N})^{\mathbb{Z}} \to (Q \times \mathbb{N})^{\mathbb{Z}}$ qui associe à toute configuration c la configuration c' telle que $c'_i = \delta(c_{i-1}, c_i, c_{i+1})$.

L'exécution d'un automate arithmétique sur une configuration initiale c consiste à itérer F_δ à partir de c jusqu'à aboutir à une configuration invariante sous l'effet de F_δ.

On peut établir maintenant des liens entre RAM arithmétiques et automates arithmétiques.

Théorème 17. Toute machine RAM arithmétique peut être simulée par un automate arithmétique.

Démonstration. Nous allons donner suffisamment d'éléments de construction pour se convaincre du résultat.

On voit une configuration $c \in (Q \times \mathbb{N})^{\mathbb{Z}}$ comme étant le ruban d'une machine RAM arithmétique. On utilise les valeurs v_i pour stocker les registres de la RAM, et les états q_i (représentés par des symboles : $\rightarrow, \nrightarrow, \cdot, PC_i...$) pour mener les calculs.

En pratique, c_0 va contenir le numéro de l'instruction en cours (*Programm Counter*), encodé par un état $q_0 = PC_i$ (s'il y a p instructions, alors on définira les états $PC_0, PC_1, ..., PC_{p-1}$). c_1, c_2, c_3 seront des registres de travail, et v_i, pour $i > 3$ correspondra au registre R_{i-3} de la RAM arithmétique que l'on simule. Les positions c_i pour $i < 0$ ne seront pas utilisées.

Lorsque l'on souhaite exécuter une instruction, on "place" une description de celle-ci dans les trois registres de travail. Ensuite, par une suite d'évolutions locales, les registres de travail vont se "déplacer" à proximité des registres $R_i, R_j...$ impliqués dans l'instruction, l'opération voulue va être appliquée, puis les registres vont être ramenés à leur position initiale.

Figure 6 est illustrée l'exécution de l'instruction $R_3 \leftarrow x$. On ne représente que les positions $c_0, c_1, c_2....$ Les configurations sont représentées les unes en dessous des autres, chaque première ligne indique les valeurs $(v_0, v_1...)$ et chaque deuxième ligne les états $(q_0, q_1...)$. Pour l'instruction "$R_3 \leftarrow x$", le registre de travail c_1 n'est pas utilisé, et initialement $v_2 = x$ et $v_3 = 3 - 1$. Lors de l'exécution, les registres de travail se positionnent au niveau de c_6 puis écrivent la valeur x dans $v_6 = R_3$ avant de revenir à leur point de départ.

L'exécution d'un instruction est terminée lorsque les registres de travail sont tous les trois dans l'état $\leftarrow$ (ou $\leftarrow^{op}$).

La figure 8 représente l'exécution de l'instruction $R_3 \leftarrow R_1 + R_3$. L'exemple n'a pas été détaillé jusqu'à son terme. En effet, on peut remarquer que la dernière configuration représentée correspond à la première configuration figure 6. Ainsi, il reste à placer la valeur $R_1 + R_3$ dans le registre R_3.

Lorsqu'une instruction a été menée à son terme, il faut charger l'instruction suivante et incrémenter le *Programm Counter*. Le nombre d'instructions constituant le programme de la machine RAM simulée étant fini, on peut encoder "directement" les instructions dans la fonction δ. Figure 5 est représenté le chargement de la $i^{ème}$ instruction : $R_3 \leftarrow x$. L'état PC_i est placé dans chacun des trois registres de travail, ce qui déclenche le chargement des états et valeurs nécessaires à l'exécution de $R_3 \leftarrow x$.

La figure 9 décrit l'exécution partielle de $R_{R_3} \leftarrow x$. Le contenue de R_3 est ramené sur le troisième registre de travail. La figure 6 illustre la suite des opérations qui amène au chargement de x dans R_{R_3}.

0	T_1	T_2	T_3	R_1	R_2	R_3
PC_i	$\leftarrow$	$\leftarrow$	$\leftarrow$	·	·	·
0	T_1	T_2	T_3	R_1	R_2	R_3
PC_i	$\leftarrow$	L	$\leftarrow$	·	·	·
0	T_1	T_2	T_3	R_1	R_2	R_3
PC_i	PC_i^-	L	$\leftarrow$	·	·	·
0	T_1	T_2	T_3	R_1	R_2	R_3
PC_i	PC_i	PC_i	$\leftarrow$	·	·	·
0	T_1	T_2	T_3	R_1	R_2	R_3
PC_{i+1}	PC_i	PC_i	PC_i	·	·	·
0	0	x	2	R_1	R_2	R_3
PC_{i+1}	$\leftarrow$	$\nleftarrow$	$\rightsquigarrow$	·	·	·

FIGURE 5 – Chargement de l'instruction suivante ($R_3 \leftarrow x$)

0	T_1	T_2	T_3	R_1	R_2	R_3
PC_i	A	A	A	·	·	·

FIGURE 7 – Etat acceptant

0	0	x	2	R_1	R_2	R_3
PC_i	$\leftarrow$	$\nleftarrow$	$\rightsquigarrow$	·	·	·
0	0	x	R_1	1	R_2	R_3
PC_i	$\leftarrow$	$\nleftarrow$	·	$\rightsquigarrow$	·	·
0	0	R_1	x	R_2	0	R_3
PC_i	$\leftarrow$	·	$\nleftarrow$	·	$\rightsquigarrow$	·
0	0	R_1	R_2	x	0	R_3
PC_i	$\leftarrow$	·	·	$\nleftarrow$	$\rightsquigarrow$	·
0	0	R_1	R_2	0	x	R_3
PC_i	$\leftarrow$	·	·	$\leftarrow$	$\uparrow$	·
0	0	R_1	0	R_2	0	x
PC_i	$\leftarrow$	·	$\leftarrow$	·	$\leftarrow$	·
0	0	0	R_1	0	R_2	x
PC_i	$\leftarrow$	$\leftarrow$	·	$\leftarrow$	·	·
0	0	0	0	R_1	R_2	x
PC_i	$\leftarrow$	$\leftarrow$	$\leftarrow$	·	·	·

FIGURE 6 – Exécution de $R_3 \leftarrow x$

On peut prolonger facilement les constructions précédentes afin de supporter toutes les instructions possibles (multiplication, goto...). Enfin, un état d'arrêt est représenté figure 7 (acceptation). Lorsqu'un tel état est en place, la configuration est invariante sous l'effet des règles d'évolution locales. □

Il reste à étudier la question de la complexité. On énonce au préalable un lemme qui garantie l'utilisation de registres d'indices "suffisamment petits" lors de l'exécution d'un programme.

Lemme 18. Toute machine RAM (resp. RAM arithmétique) R peut-être simulée en temps polynomial par une machine RAM (resp. RAM arithmétique) R' telle qu'au temps T de l'exécution de R' le plus grand indice de registre utilisé est un $O(T)$.

Démonstration. Si lecture et chargement indirects n'étaient pas permis, il suffirait de renuméroter les registres intervenant dans le programme. Les opérations indirectes nous obligent à adopter une autre démarche.

La solution consiste à ne plus stocker la valeur de chaque registre R_i à la position i de la mémoire, mais à enregistrer (sur les deux premiers registres consécutifs disponibles) le couple (i, R_i). Ainsi, les registres ayant une valeur assignée sont disponibles en début de mémoire.

Lorsqu'une instruction souhaite accéder en lecture au $i^{ème}$ registre de valeur R_i, il lui suffit de repérer le couple (i, R_i). Lors de l'écriture d'une valeur R sur le $i^{ème}$ registre, on vérifie si un couple de la forme $(i, \_)$ est déjà présent. Si c'est le cas, on le remplace par (i, R), sinon on ajoute (i, R) sur les deux premiers registres libres.

Le schéma ci-dessous illustre en exemple l'état de la mémoire à l'issu de l'exécution de : $V_2 \leftarrow 100$; $V_{V_2} \leftarrow 1$.

Numéro de registre	1	2	3	4	5	6
Contenu	2	100	100	1	0	0

Il est facile de constater que l'on définit ainsi une machine R' capable de simuler en temps polynomial la machine R initiale, et telle qu'au temps T de son exécution, le plus grand indice de registre utilisé est un $O(T)$. □

0	1	2	2	R_1	R_2	R_3
PC_i	$\rightarrow^+$	$\downarrow$	$\rightarrow^+$	·	·	·
0	2	0	R_1	1	R_2	R_3
PC_i	$\downarrow$	$\rightarrow^+$	·	$\rightarrow^+$	·	·
0	2	R_1	R_1	R_2	0	R_3
PC_i	$\downarrow$	$\leftarrow^+$	·	·	$\rightarrow^+$	·
0	R_1	2	R_1	R_2	R_3	R_3
PC_i	$\leftarrow^+$	$\curvearrowright$	·	·	$\leftarrow^+$	·
0	R_1	2	R_1	R_3	R_2	R_3
PC_i	$\leftarrow^+$	$\curvearrowright$	·	$\leftarrow^+$	·	·
0	R_1	2	R_3	R_1	R_2	R_3
PC_i	$\leftarrow^+$	$\curvearrowright$	$\leftarrow^+$	·	·	·
0	R_1	$R_1 + R_3$	2	R_1	R_2	R_3
PC_i	$\leftarrow^+$	$\twoheadrightarrow$	$\rightsquigarrow$	·	·	·

FIGURE 8 – Exécution (partielle) de $R_3 \leftarrow R_1 + R_3$

0	0	x	2	R_1	R_2	R_3
PC_i	$\leftarrow$	$\downarrow$	$\rightarrow^R$	·	·	·
0	0	x	R_1	1	R_2	R_3
PC_i	$\leftarrow$	$\downarrow$	·	$\rightarrow^R$	·	·
0	0	x	R_1	R_2	0	R_3
PC_i	$\leftarrow$	$\downarrow$	·	·	$\rightarrow^R$	·
0	0	x	R_1	R_2	R_3	R_3
PC_i	$\leftarrow$	$\downarrow$	·	·	$\leftarrow^R$	·
0	0	x	R_1	R_3	R_2	R_3
PC_i	$\leftarrow$	$\downarrow$	·	$\leftarrow^R$	·	·
0	0	x	R_3	R_1	R_2	R_3
PC_i	$\leftarrow$	$\downarrow$	$\leftarrow^R$	·	·	·
0	0	x	R_3	R_1	R_2	R_3
PC_i	$\leftarrow$	$\twoheadrightarrow$	$\rightsquigarrow$	·	·	·

FIGURE 9 – Exécution (partielle) de $R_{R_3} \leftarrow x$

On obtient enfin le théorème suivant, qui nous permettra par la suite d'utiliser des automates arithmétiques à la place des RAM arithmétiques :

Théorème 19. Toute machine RAM arithmétique peut être simulée en temps polynomiale par un automate arithmétique.

Démonstration. Considérons une machine RAM arithmétique R s'exécutant en temps $T(n)$. D'après le lemme 18, il existe une machine RAM arithmétique R' et un polynôme Q tel que R' s'exécute en temps $Q(T(n))$ et tel que le plus grand indice de registre utilisé est un $O(Q(T(n)))$. Par conséquent, toute instruction intervenant dans le programme de R' peut être simulée en temps $O(Q(T(n)))$ par un automate arithmétique. R peut donc être simulée en temps polynomial par un automate arithmétique. □

4 Caractérisation de PSPACE par machines discrètes

Afin d'obtenir une caractérisation de PSPACE par équations différentielles, il nous faut au préalable en obtenir une par machines discrètes. Nous reprenons ici les démonstrations prouvant que les machines à Vecteurs et les machines RAM arithmétiques reconnaissent les langages de PSPACE en temps polynomial.

4.1 Caractérisation par Machines à Vecteurs : la thèse du calcul parallèle

Il existe plusieurs modèles de calcul parallèle, tels que les PRAM (Parallel Random Access Machine) ou les APM (Array Processing Machines). Une description détaillée de ceux-ci est disponible dans [BDG90]. Nous nous intéressons ici plus particulièrement aux machines à Vecteurs. Les opérations bit à bit supportées par ces dernières s'assimilent en effet à du calcul parallèle.

L'intérêt que nous portons aux machines de calcul parallèle s'explique par l'existence d'une thèse, analogue à celle de Church concernant les machines de Turing, vérifiée entre autres par les machines à Vecteurs.

Théorème 20 (Thèse du calcul parallèle)**.** L'ensemble des langages reconnus en temps polynomial sur un modèle de calcul parallèle raisonnable est égale à PSPACE.

Nous donnons ci-dessous quelques éléments de preuve (issus de [BDG90] et [PS76]) permettant d'affirmer que les machines à Vecteurs vérifient la thèse du calcul parallèle. On notera P_{MV} les langages reconnus en temps polynomial par machine à Vecteurs.

Théorème 21 ([PS76])**.** L'inclusion suivante est vérifiée : $P_{MV} \subseteq PSPACE$

Démonstration. Soit $\mathcal{L} \in P_{MV}(T(n))$ (tout mot de longueur n est reconnu en temps $T(n)$ où T et un polynôme). Considérons une machine RAM avec un programme P acceptant $\mathcal{L}$ en temps $T(n)$. On rappelle qu'il existe deux constantes p et q tel que la taille de toute index est bornée par $p+t$ au temps t, et la taille de tout registre par $2^{p+t}+q$.

On souhaite simuler l'exécution de P sur une machine de Turing utilisant un espace polynomial en la taille de l'entrée. On peut remarquer que seules les instructions `goto` nécessitent de connaître la valeur d'un registre afin de déterminer l'instruction suivante à exécuter. De plus, il s'agit uniquement d'un test d'égalité à 0. On considère alors la fonction $find(b,i,t)$ qui retourne le $b^{ème}$ bit du registre R_i au temps t. Le test $V_i = 0$ est donc équivalent à : $\bigvee_{i=1}^{2^{p+t}+q} find(b,i,t)$.

On construit une machine de Turing qui enregistre successivement les couples (i_j, j) où i_j indique le numéro de l'instruction exécutée au temps j : $(i_1,1),(i_2,2),(i_3,3)$.... Lorsqu'une instruction avec `goto` est rencontrée, on utilise la fonction $find$ pour déterminer la suite du programme à exécuter. Il suffit donc de pouvoir évaluer la fonction $find$ en espace polynomial en la taille de l'entrée. Etudions les différents cas possibles lors de l'évaluation de $find(b,i,t)$:

- si $t = 1$, parcourir directement R_i
- si l'instruction exécutée au temps t ne change pas la valeur de R_i, renvoyer $find(b,i,t-1)$
- si l'instruction au temps t est $R_i \leftarrow x$, parcourir directement x
- si l'instruction au temps t est $R_i \leftarrow R_j \wedge R_k$ (resp. $R_i \leftarrow \neg R_j$), renvoyer $find(b,j,t-1) \wedge find(b,k,t-1)$ (resp. $\neg find(b,j,t-1)$)
- si l'instruction au temps t est $R_i \leftarrow R_j \uparrow I_k$. On sait que I_k est borné par $p+t$. On peut obtenir facilement la valeur de l'index I_k (par une procédure que l'on ne décrit pas ici, voir [BDG90]). Supposons $I_k > 0$. Si $b - I_k < 0$ alors $find(b,i,t) = 0$ (le shift a rempli la position b par un 0), si $0 \leq v - I_k \leq 2^{p+t}+q$ alors $find(b,i,t) = find(b-I_k,i,t-1)$, si $b - I_k > 2^{p+t}+q$ alors $find(b,i,t) = find(2^{p+t}+q+1,i,t-1)$ (on cherche le signe du nombre). Les autres cas se traitent de manière similaire (voir [BDG90]).

Le stockage de la liste $(i_1,1),(i_2,2),(i_3,3)...(i_{T(n)},T(n))$ nécessite une place $O(T(n)\cdot \log T(n))$. Tout registre étant bornée par $O(2^{p+t}+q)$, leur taille est en $O(T(n)+\log n)$. La fonction $find$ peut être implémentée avec une pile de profondeur $O(T(n))$, chaque niveau nécessitant une place en $O(T(n)+\log n)$. On obtient donc bien le résultat voulu. □

Théorème 22 ([PS76])**.** L'inclusion suivante est vérifiée : $PSPACE \subseteq P_{MV}$

Démonstration. On démontre que $NPSPACE \subseteq P_{MV}$, le théorème de Savitch ($PSPACE = NPSPACE$) permet de conclure.

Soit $\mathcal{L} \in NPSPACE(S(n))$. On considère une machine de Turing à deux rubans reconnaissant $\mathcal{L}$. Le premier ruban contient $\$w\$$ où w est le mot (de taille n) donné en entrée, et n'est accessible qu'en lecture. Le second est un ruban de travail accessible en lecture et en écriture. On encode une configuration de la machine par le mot $0^{n-j+1}10^j\tau_1 q \tau_2$. j indique la position de la tête de lecture sur le premier ruban, q désigne l'état de la machine, τ_2 la parti du ruban située à droite de la deuxième tête de lecture, et τ_1 la partie située à gauche. On a $|w| = n$ et $|\tau_1 q \tau_2| = S(n)$ (les caractères blancs sont encodés par #). On note γ_1, γ_2... toutes les configurations possibles (dont certaines sont irréalisables en pratique).

Il est possible de construire par machine à Vecteurs un vecteur γ constitué de la concaténation de toutes les configurations en temps $O(S(n)+\log n)$ (voir [BDG90]). Une seconde machine à Vecteurs travaillant ensuite sur γ peut produire en temps $O(S(n)+\log n)$ une matrice $Step$ (représentée sous forme d'un vecteur) tel que $Step(i,j)$ vaut 1 si et seulement si γ_j est une configuration pouvant succéder à γ_i. Afin d'effectuer cette tâche rapidement, la machine à Vecteurs tire parti des opérations bit à bit pour traiter *en parallèle* tous les couples de configurations possibles.

Enfin, la clôture transitive de la matrice $Step$ s'obtient en temps $O((S(n)+\log n)^2)$ (voir [BDG90]). La valeur de $Step(Init, Acc)$ (où $Init$ est le numéro de la configuration initiale, et Acc celui de la configuration acceptante) indique si le mot w donné en entrée appartient ou non à $\mathcal{L}$. □

4.2 Caractérisation par Machines RAM arithmétiques

Bien que cela puisse surprendre au premier abord, les opérations $\times$ et $\div$ confèrent une puissance supplémentaire non négligeable aux machines RAM. En effet, il a été prouvé dans [BMS81] que

les RAM arithmétiques calculent PSPACE en temps polynomial. Nous allons présenté ce résultat ci-dessous.

Définition 23. Une QBF (*Quantified Boolean Formula*) ψ est une formule logique du premier ordre de la forme :

$$\psi = Q_1\ x_1\ Q_2\ x_2 \cdots Q_n\ x_n\ \phi(x_1, ..., x_n)$$

où $Q_i \in \{\forall, \exists\}$ et ϕ est une formule propositionnelle construite à partir des variables $x_1, ..., x_n$.

Le problème QBF est PSPACE-complet. Nous allons nous intéresser à un problème plus complexe à première vue : déterminer le nombre d'assignations acceptantes d'une QBF.

Définition 24. Soit ψ une QBF, on note $\#\psi$ le nombres d'assignations acceptantes de ψ. On note #QBF le problème consistant à trouver le nombre d'assignations acceptantes d'une QBF.

Exemple 25. On ne définit pas précisément la notion d'assignation dans le cas d'une #QBF (se référer à [BMS81]). Considérons tout de même un exemple avec la QBF $\psi = \exists x_1 \forall x_2 \exists x_3 (x_1 \wedge (x_1 \vee x_2 \implies x_3))$. Cette QBF comporte 8 assignations différentes. En voici 3 possibles : $x_1 = 0$, $x_3 = 0$ lorsque $x_2 = 0$ et $x_3 = 0$ lorsque $x_2 = 1$ | $x_1 = 0$, $x_3 = 0$ lorsque $x_2 = 0$ et $x_3 = 1$ lorsque $x_2 = 1$ | $x_1 = 1$, $x_3 = 0$ lorsque $x_2 = 0$ et $x_3 = 0$ lorsque $x_2 = 1$. Seule l'assignation suivante est acceptante : $x_1 = 1$, $x_3 = 1$ lorsque $x_2 = 0$ et $x_3 = 1$ lorsque $x_2 = 1$.

Si l'on parvient à résoudre toute instance de #QBF en temps polynomial avec une RAM arithmétique, alors il en va de même pour QBF (une QBF est satisfiable si et seulement si son nombre d'assignations acceptantes est non nul). Nous allons donc démontrer comment $\#\psi$ peut être calculé en temps polynomial par une RAM arithmétique. Nous allons définir pour ce faire une structure particulière conduisant à l'utilisation de *straight line programs* (notés *slp* par la suite).

Définition 26. Pour tout polynômes f à coefficients entiers, on note f_k son coefficient de rang k.
Soit f et g deux polynômes à coefficients entiers. On définit les opérations suivantes :

Opération	Notation	Définition
Addition	$f + g$	$(f+g)_k = f_k + g_k$
Multiplication	$f \cdot g$	$(f \cdot g)_k = \sum_{i=0}^{k} f_j \cdot c_{k-j}$
Décalage	$\|f$	$(\|f)_k = f_{k+1}$
Multiplication d'Hadamard	$f \otimes g$	$(f \otimes g)_k = f_k \cdot g_k$
Extraction	$\boxed{\text{h}}f$ (où $h \geq 2$ est un entier)	$(\boxed{\text{h}}f)_k = f_{h \cdot k}$

On note $\mathcal{P} = (\mathbb{P}, +, \cdot, |, \otimes, \boxed{\text{h}})$ la structure constituée de l'ensemble $\mathbb{P}$ des polynômes à coefficients entiers, muni des opérations définies précédemment.

Exemple 27. Considérons les polynômes $f = 3 + 4x + x^2 + 8x^3 + 7x^6$ et $g = x + 3x^3$. On a $f \cdot g = 3x + 4x^2 + 10x^3 + 12x^4 + 3x^5 + 24x^6 + 7x^7 + 21x^9$, $|f = 4 + x + 8x^2 + 7x^5$, $f \otimes g = 4x + 24x^3$ et $\boxed{3}f = 3 + 8x + 7x^2$.

Définition 28. Un *slp* de taille n sur la structure $\mathcal{P}$ est une succession de n instructions. La première est :

$$(1)\ \ p_1 \leftarrow x \ \ (x \text{ désigne le polynôme élémentaire})$$

Ensuite chaque instruction k est de la forme :

$$(k)\ \ p_k \leftarrow p_i + p_j\ ,\ p_i \cdot p_j\ ,\ |p_i\ ,\ p_i \otimes p_j\ ,\ \boxed{\text{h}}p_i\ ,\ 1$$
$$(\text{où } i, j < k \text{ et } 1 \text{ désigne le polynôme constant})$$

Etant donné un *slp* Π de taille n, on dit que Π génère le polynôme p_n.

On montre comment associer à toute QBF ψ un *slp* de taille polynomial en $|\psi|$ qui génère un polynôme constant égal à $\#\psi$.

Théorème 29 ([BMS81])**.** Soit ϕ une formule propositionnelle comportant n variables $x_1, ..., x_n$ et m connecteurs logiques. On définit le polynôme $p_\phi = \sum_{k=0}^{2^n-1} a_k \cdot x^k$ où $a_k = 1$ si, en assignant à chaque x_i la valeur du $i^{ème}$ bit de k, la formule ϕ est satisfaite, 0 sinon.

Le polynôme p_ϕ est calculable par un *slp* de taille $O(n+m)$.

Démonstration. Il suffit de faire les constatations suivantes :

— $p_\zeta = \sum_{k=0}^{2^n-1} x^k = \prod_{k=1}^{n-1}(1+x^{2^k})$ (où ζ désigne la formule propositionnelle toujours vraie)

— $p_{x_j} = (1+x)\cdots(1+x^{2^j-1})\cdot x^{2^j}\cdot(1+x^{2^j+1})\cdots(1+x^{2^{n-1}}) = \frac{x^{2^j}}{1+x^{2^j}}\cdot p_\zeta$

— $p_{\alpha\vee\beta} = p_\alpha + p_\beta - p_\alpha \otimes p_\beta$

— $p_{\alpha\wedge\beta} = p_\alpha \otimes p_\beta$

— $p_{\neg\alpha} = p_\zeta - p_\alpha$

La preuve s'établit ensuite par induction structurelle sur ϕ. □

Théorème 30 ([BMS81])**.** Soit $\psi = Q_1x_1Q_2x_2\cdots Q_nx_n\phi(x_1,...,x_n)$ une QBF (où $Q_i \in \{\forall, \exists\}$ et ϕ comporte m connecteurs logiques). Il existe un *slp* de taille $O(n+m)$ générant un polynôme constant égale à $\#\psi$.

Démonstration. On a :

$$\#\psi = O_1O_2...O_n\phi(x_1,...,x_n)$$

où :

$$O_i = \prod_{x_i\in\{0,1\}} \text{ si } Q_i = \forall \text{ et } O_i = \sum_{x_i\in\{0,1\}} \text{ si } Q_i = \exists$$

Le *slp* suivant produit alors bien le polynôme constant $\#\psi$:

— $p_0 = p_\phi$ (p_ϕ s'obtient par le *slp* de taille $O(n+m)$ définit théorème 29)

Pour $0 < j \leq n$:

— $p_j = \boxed{2}(p_{j-1} \otimes (|p_{j-1}))$ si $Q_{n-j+1} = \forall$

— $p_j = \boxed{2}(p_{j-1} + (|p_{j-1}))$ si $Q_{n-j+1} = \exists$

(si $p = a_0 + a_1x + a_2x^2 + \cdots$ alors $\boxed{2}(p \otimes (|p)) = a_0a_1 + a_2a_3x + a_4a_5x^2$ et $\boxed{2}(p + (|p)) = (a_0+a_1) + (a_2+a_3)x + (a_4+a_5)x^2 + \cdots$) □

Il reste enfin à simuler des *slp* avec des RAM arithmétiques.

Théorème 31 ([BMS81])**.** Soit Π un *slp* de taille n, soit $x \geq 2$ un entier. Il est possible de calculer $p_n(x)$ en $O(n^2)$ opérations avec une RAM arithmétique.

Démonstration. Le résultat s'obtient par induction structurelle. En effet, on a :

— $(f+g)(x) = f(x) + g(x)$

— $(f\cdot g)(x) = f(x)\cdot g(x)$

— $|f(x) = f(x)/x$

L'opération $\boxed{\text{h}}$ est plus complexe à obtenir. Considérons pour ce faire y très grand devant x et les coefficients de f (se référer à [BMS81] pour plus de détails). Supposons que f est de degré p et notons (q_k) et (r_k) les suites d'entiers telles que pour tout $k \leq p$: $k = q_k\cdot h + r_k$ et $(r_k < h)$. On a alors :

$$\begin{aligned}(f(y) \bmod y^h - x) \bmod y &= \left(\sum_{k=0}^{p} f_k \cdot [(y^{q_k\cdot h} - x^{q_k})\cdot y^{r_k} + x^{q_k}\cdot y^{r_k}] \bmod y^h - x\right) \bmod y \\ &= \left(\sum_{k=0}^{p} f_k \cdot x^{q_k}\cdot y^{r_k} \bmod y^h - x\right) \bmod y \\ &= \sum_{r_k=0} f_k \cdot x^{q_k} \bmod y = \boxed{\text{h}} f(x)\end{aligned}$$

On démontre de manière similaire que :

$$(f\otimes g)(x) = (f(y)\cdot g(y^{n+1}) \bmod y^{n+2} - x) \bmod y$$

□

L'ensemble des résultats précédents nous permet enfin d'énoncer le théorème recherché.

Théorème 32 ([BMS81])**.** L'ensemble des langages reconnus en temps polynomial par machine RAM arithmétique est exactement PSPACE.

Le corollaire suivant, obtenu à partir des théorèmes 19 et 32, transpose le résultat précédent aux automates arithmétiques.

Corollaire 33. Tout langage de PSPACE est reconnu en temps polynomial par automate arithmétique.

5 Caractérisation de PSPACE par équations différentielles

On adopte ici une nouvelle démarche visant à caractériser PSPACE, différente de celle exposée partie 2.4. Nous allons sortir du cadre des équations différentielles ordinaires, propre au GPAC, pour essayer de simuler une des machines présentées en partie 3.

5.1 Problématique de simulation

Nous avons vu une première caractérisation de PSPACE partie 2.4 en utilisant des oracles. Toutefois, cette façon d'obtenir PSPACE est quelque part très artificielle, et consiste essentiellement à ajouter la possibilité de résoudre les problèmes de PSPACE au modèle. Nous allons donc adopter une autre démarche afin d'obtenir un résultat plus "naturel".

Trois machines ont été présentées partie 3, les machines à Vecteurs, les RAM arithmétiques et les automates arithmétiques. Celles-ci ont pour particularité de reconnaître les langages de PSPACE en temps polynomial. Par conséquent, si nous parvenons à simuler l'une d'entre elles avec une équation différentielle, nous aurons bien obtenu une nouvelle caractérisation de PSPACE par équations différentielles.

Les machines à Vecteurs ont été le premier modèle à retenir notre attention. Une des caractéristiques de ce modèle est l'utilisation d'un nombre fini de registres, prédéterminé par le programme exécuté (ceci est rendu possible par l'absence d'adressage indirect). Ainsi, il est possible de définir la configuration d'une machine à Vecteurs comme étant un vecteur de dimension finie $c = (v_1, v_2 ... v_p, i)$ où les v_j sont les valeurs des registres utilisés et i est le numéro de l'instruction en cours (*Programm Counter*). Ceci nous rapproche du travail effectué dans [BGP14], et présenté partie 2.3, où, pour simuler une machine de Turing, l'on définit une configuration par un vecteur de dimension 5.

Il reste ensuite à construire une fonction $step : \mathbb{Z}^{p+1} \to \mathbb{Z}^{p+1}$ tel que $step(c)$ donne la configuration succédant à c. $step$ doit faire appel à des fonctions que l'on pourra prolonger sur $\mathbb{R}$ et peu sensibles aux perturbations, comme exposé partie 2.3. Si le programme P considéré comporte s instructions, on peut définir pour chaque instruction j une fonction $step_j$ tel que $step_j(c)$ applique l'instruction j à la configuration c. En notant $\mathbb{1}$ la fonction qui vaut 1 en 0 et 0 partout ailleurs, on a alors :

$$step(c) = step\begin{pmatrix} v_1 \\ v_2 \\ ... \\ v_p \\ i \end{pmatrix} = \sum_{j=1}^{s} \mathbb{1}(i-j) \cdot step_j(c)$$

Il semble aisé de construire les fonctions $step_j$ pour les instructions de chargement direct, de négation, de shift et de `goto`. L'opération de conjonction bit à bit est plus problématique. En effet, il faudrait une fonction $conj : \mathbb{Z}^2 \to \mathbb{Z}$ qui effectue la tâche demandée, et que l'on puisse prolonger sur $\mathbb{R}$ en une fonction $\overline{conj} : \mathbb{R}^2 \to \mathbb{R}$ telle que $\overline{conj}(n + \varepsilon)$ soit suffisamment proche de $\overline{conj}(n)$ (stabilité). Une idée fut de coder chaque nombre en intercalant des 0 une position sur deux (par exemple, $5 = (101)_2$ est encodé par 10001). Ceci permet *presque* d'obtenir la disjonction bit à bit lorsque l'on somme deux nombres (par exemple, $5 \vee 3 = (101)_2 \vee (11)_2 = (111)_2$ et $10001 + 101 = 10110$). Il resterait à trouver une fonction de correction qui décalerai d'un cran vers

la droite les éventuels 1 apparus un bit sur deux (par exemple, $correct(10110) = 10101$), nous n'y sommes pas parvenu.

Il a donc fallu trouver un autre modèle de calcul, ne faisant pas appel aux opérations bit à bit. Le modèle RAM arithmétique rempli la tâche demandée, cependant, il n'est pas utilisable en l'état. En effet, ce modèle peut nécessiter l'usage d'un nombre non borné de registres, du fait de l'adressage indirect. Définir une configuration comme étant un vecteur de dimension infinie n'est pas convenable, puisque cela conduirait à des équations différentielles de dimension infinie. Afin de palier à ce problème, nous avons construit un modèle de calcul équivalent aux RAM arithmétiques mais tel que la valeur de chaque registre ne dépende plus que de celle de ses deux registres adjacents, c'est le modèle des automates arithmétiques. L'intuition est alors que les automates arithmétiques peuvent être simulés par des équations différentielles de la forme :

$$\frac{\partial y(t,x)}{\partial t} = p(y(t,x-1), y(t,x), y(t,x+1))$$

En effet, dans la simulation des machines de Turing par GPAC, $y(n)$, pour n entier, indique la configuration de la machine après n itérations. Ici, $y(n,v)$ représente la valeur du registre v après n itérations, et dépend uniquement de $y(n,v-1)$, $y(n,v)$ et $y(n,v+1)$. Nous allons développer cette idée dans la partie suivante.

5.2 Simulation des automates arithmétiques par équations différentielles

Nous allons esquisser ici plus en détail les résultats permettant de simuler des automates arithmétiques par équations différentielles. Bien que la caractérisation recherchée reste encore à l'état de conjecture (voir ci-dessous), nous pensons que les résultats qui suivent sont les préliminaires indispensables à sa démonstration. La démarche adoptée s'apparente à celle déjà utilisée avec les machines de Turing (voir partie 2.3 et [BGP14]).

La première étape consiste à définir un encodage des nombres que l'on va manipuler. En effet, si l'on considère l'encodage naturel $\psi(w) = \sum_{i=1}^{n} w_i 2^i$ (sur $\{0,1\}$), alors $\|\psi(w)\| \approx 2^{|w|}$. Or l'on souhaite exploiter l'inégalité $\text{length}(y)([0,t]) \leqslant t\,\text{poly}(\sup_{u\in[0,t]} \|y(u)\|)$ pour garantir que les longueurs de courbes soient polynomiales (et donc obtenir un temps de calcul polynomial par équations différentielles, comme présenté définition 4), ce qui nécessite que $||\psi(w)||$ soit polynomial en $\|w\|$. L'encodage proposé dans [BGP14] (et présenté partie 2.3) est bien polynomial, mais il ne permet pas d'effectuer d'opérations arithmétiques (les retenus se propagent dans le mauvais sens). Nous avons donc orienté nos recherches vers un nouvel encodage qui représente chaque nombre x par un couple $\langle x\rangle = (X, n_x) = (\frac{x}{2^{n_x}}, n_x)$ où n_x doit vérifier $n_x \geq |x|$. Les opérations arithmétiques deviennent alors possible :

$$\langle x+y\rangle = (X\cdot 2^{-n_y} + Y\cdot 2^{-n_x}, n_x+n_y) \qquad \left\langle \frac{x}{y}\right\rangle = \left(\frac{X}{Y}, n_x - n_y + 1\right)$$

$$\langle x-y\rangle = (X\cdot 2^{-n_y} - Y\cdot 2^{-n_x}, n_x+n_y) \qquad \langle x\cdot y\rangle = (X\cdot Y, n_x+n_y)$$

Toutefois, il faut contrôler la valeur de n_x qui peut devenir trop importante (du fait par exemple de l'addition ou de la soustraction avec les calculs ci-dessus). L'idéal serait de pouvoir conserver n_x très proche de $|x|$ tout au long des calculs, nous n'avons pas encore trouvé de moyen d'effectuer cela.

Il est nécessaire ensuite de définir une fonction $\overline{\delta} : (\mathbb{R}^2)^3 \to \mathbb{R}^2$ qui prolonge et approxime la fonction $\delta : (Q\times\mathbb{N})^3 \to Q\times N$ d'évolution locale de l'automate que l'on simule. Avant d'évoquer cela, nous énonçons ci-dessous un lemme central qui permettra d'itérer $\overline{\delta}$ à partir d'une configuration initiale c, jusqu'à entrer dans un état d'arrêt. Ce sont les conditions d'application de ce lemme qui restreignent la construction de $\overline{\delta}$.

Définition 34. Soit $F : (\mathbb{R}^2)^3 \to \mathbb{R}^2$ et $p : \mathbb{R}^2 \to \mathbb{R}$ un polynôme positif. Définissons pour $i \in \{1,2\}$ le système d'équations différentielles suivant :

$$\begin{cases} \frac{\partial z_i(t,x)}{\partial t} = A_i s_{1/8}(t, B_i) X_3(F_i(u(t,x-1), u(t,x), u(t,x+1)) - z_i) \\ \frac{\partial u_i(t,x)}{\partial t} = C_i s_{1/8}(t-1/2, D_i) X_3(z_i - u_i) \end{cases} \tag{2}$$

où :

$$\begin{cases} A_i = 4 + 8p(u) \\ B_i = 2p(z) + \log A_i(1 + X_3(F_i(u(t,x-1), u(t,x), u(t,x+1)) - z_i)^2) \\ C_i = 4 + 8p(z) \\ D_i = 2p(u) + \log C_i(1 + X_3(z_i - u_i)^2) \\ X_3(y) = y + y^3 \end{cases}$$

Lemme 35. Soit $F : (\mathbb{R}^2)^3 \to \mathbb{R}^2$ et $(x_j)_j \in \mathbb{R}^{2^{\mathbb{Z}}}$. On note $(x_{1,j}, x_{2,j}) = x_j$ et $\tau_j = (x_{j-1}, x_j, x_{j+1})$. Supposons que z et u soient solutions de (2). Considérons $M \in \mathbb{R}$ et $\delta, \Delta, \varepsilon \in \mathbb{R}^2$ tels que pour tout $i \in \{1,2\}, j \in \mathbb{Z}$:

$$|u_i(0,j) - x_{i,j}|, |z_i(0,j) - x_{i,j}| \leq \delta_i$$
$$p(x_j + \varepsilon), p(F(\tau_j) + \varepsilon) \geq M$$
$$|F_i(\tau_j + (\varepsilon, \varepsilon, \varepsilon)) - F_i(\tau_j)| \leq \Delta_i$$

où :

$$\varepsilon_i \leq max(\delta_i, \Delta_i) + 2e^{-M} \qquad \varepsilon_i \leq \delta_i + 2e^{-M}$$

Alors :

$$|u_i(1,j) - F_i(\tau_j)|, |z_i(1,j) - F_i(\tau_j)| \leq \Delta_i + 2e^{-M}$$

Démonstration. Ce lemme est une généralisation du lemme 64 exposé dans [BGP14]. La démonstration peut s'obtenir ici aisément en fixant le paramètre j et en constatant que l'on retrouve alors les conditions d'application de [BGP14]. Nous ne donnerons pas de détails techniques concernant ce lemme ou la définition 34, à la place, nous exposons ci-dessous quelques éléments d'interprétation dans le contexte des automates arithmétiques.

La fonction F s'assimile à la fonction $\overline{\delta}$ que l'on souhaite itérer. La suite $(x_j)_j$ désigne la configuration initiale de l'automate, et τ_j un triplet de cases adjacentes. Ainsi, $F(\tau_j)$ correspond à l'application de la fonction locale d'évolution de l'automate sur la position x_j. L'objectif est de faire évoluer simultanément l'ensemble des x_j, sous l'action de F, jusqu'à aboutir à une configuration stable. Ceci correspondrait à la simulation d'un automate arithmétique.

Les fonctions u et z représentent en pratique les itérations successives de F. Leur premier argument s'apparente au temps, et leur second à une position de l'automate. Ainsi, $u(n,x)$ et $z(n,x)$ (pour n un entier positif et x un entier relatif) sont des approximations de x_j après n itérations de la fonction d'évolution de l'automate à partir de la configuration initiale. La condition $|u_i(0,j) - x_{i,j}|, |z_i(0,j) - x_{i,j}| \leq \delta_i$ signifie qu'initialement (au temps 0), $u_i(0,j)$ et $z_i(0,j)$ sont déjà de bonnes approximations de la configuration initiale.

La condition $|F_i(\tau_j + (\varepsilon, \varepsilon, \varepsilon)) - F_i(\tau_j)| \leq \Delta_i$ indique que la fonction que l'on souhaite itérer est relativement peu sensible aux perturbations. C'est cette condition qui nous oblige à rechercher de bonnes approximations lorsque l'on construit la fonction $\overline{\delta}$.

Sous l'ensemble des conditions évoquées dans le lemme, on obtient ($|u_i(1,j) - F_i(\tau_j)|, |z_i(1,j) - F_i(\tau_j)| \leq \Delta_i + 2e^{-M}$). Ceci signifie qu'après un pas de calcul, $u_i(1,j)$ et $z_i(1,j)$ sont de bonnes approximations de $F(\tau_j)$. On peut alors ré-appliquer le lemme à $u_i(1,j)$ et $z_i(1,j)$ pour démontrer que $u_i(2,j)$ et $z_i(2,j)$ sont à nouveaux de bonnes approximations de $F^{[2]}(\tau_j)$, et ainsi de suite... □

Venons-en à la fonction $\overline{\delta}$. Le lemme précédant nous oblige à définir une fonction $\overline{\delta}$ suffisamment peu sensible au perturbations pour que les erreurs induites au cours de la simulation demeurent raisonnables. La définition des automates arithmétiques (voir définition 15 et notations associées), ainsi que le procédé d'interpolation définit en partie 2.3 nous conduisent tout naturellement à la fonction $\overline{\delta} : (\mathbb{R}^2)^3 \to \mathbb{R}^2$ suivante :

$$\overline{\delta}(x_{j-1}, x_j, x_{j+1}) = (L_{\delta_Q}(\overline{\Gamma}), \sum_{i=1}^{s} L_{\delta_{\mathbb{N},i}}(\overline{\Gamma}) \cdot A_i)$$

où :

- $\overline{\Gamma} = (x_{1,j-1}, x_{1,j}, x_{1,j+1}, L_{\mathbb{1}}(x_{2,j-1}), L_{\mathbb{1}}(x_{2,j}), L_{\mathbb{1}}(x_{2,j+1}))$ ($\mathbb{1}$ vaut 1 en 0 et 0 partout ailleurs)
- $A_i = x$ ou $A_i = x\ op\ y$ avec $x, y \in \{1, v_0, v_1, v_2\}$ et $op \in \{+, -, \times, \div\}$
- L_f désigne l'interpolation de Lagrange de f (voir partie 2.3)

— $x_{1,j}$ représente q_j (état de la position j de l'automate) et $x_{2,j}$ représente v_j (valeur de la position j)

Malheureusement cette fonction ne remplit pas les conditions d'application du lemme 35. On pourrait reprendre certaines fonctions de [BGP14] pour améliorer l'approximation (par exemple, remplacer $L_{\delta_{\mathbb{N},i}}(\overline{\Gamma})$ par $\overline{int}(L_{\delta_{\mathbb{N},i}}(\overline{\Gamma}))$ où $\overline{int}$ est une approximation de la partie entière), sans toutefois atteindre les bornes voulues. Il manque en l'état aux automates arithmétiques une technique similaire au Zig-Zag pour les machines de Turing. Rappelons que le Zig-Zag consiste à utiliser une propriété de la machine simulée pour corriger fréquemment les composantes de la configuration les plus perturbées (dans le cas des machines de Turing Zig-Zag, la machine atteint régulièrement un bord du ruban).

Il demeure donc à adapter le modèle des automates arithmétiques pour répondre aux critères du lemme 35. Ceci conclurait la démonstration de la conjecture ci-dessous, qui produit une caractérisation de la classe PSPACE par équations différentielles plus élégante et plus raisonnable que le théorème 7 :

Conjecture 36. PSPACE est inclue dans l'ensemble des langages reconnus en temps polynomial (au sens de la définition 4) par une équation différentielle de la forme :

$$\frac{\partial y(t,x)}{\partial t} = p(y(t,x-1), y(t,x), y(t,x+1))$$

où $p : \mathbb{R}^{d^3} \to \mathbb{R}^d$ est un vecteur de polynômes.

6 Conclusion

Nous avons présenté dans ce rapport plusieurs tentatives de caractérisation de la classe PSPACE par équations différentielles à second membre polynomial. La première caractérisation (théorème 7) fait appel à la notion d'oracle, mais s'avère être trop artificielle. Nous avons donc recherché un modèle de calcul discret qui reconnaisse la classe PSPACE en temps polynomial, ce qui a conduit à la construction d'un nouveau modèle : les automates arithmétiques. Bien que la simulation de ces derniers par équations différentielles n'a pas été menée à terme, la conjecture 36 présente le résultat attendu et dont nous avons donné des éléments de preuve. Cette dernière caractérisation, plus élégante que la précédente, laisse entrevoir pour la classe PSPACE et les équations différentielles des résultats similaires à ceux établis dans [BGP14].

La démonstration de la conjecture 36 nécessite encore du travail. La piste la plus probante nous semble être celle exposée partie 5.2, en utilisant des automates arithmétiques. Par ailleurs, la possibilité de simuler des machines à vecteurs, malgré les problèmes présentés partie 5.1, reste également une question ouverte.

Enfin, le travail et les méthodes exposés dans ce rapport pourraient être utiles dans le futur afin d'étendre les résultats de calculabilité et de complexité concernant les équations différentielles à second membre polynomial. Il est par exemple envisageable de simuler d'autres machines discrètes aux propriétés calculatoires particulières, à l'aide d'équations différentielles, pour caractériser d'autres classes de complexité. Un objectif à terme serait de reproduire la hiérarchie polynomiale du point de vue des équations différentielles.

Références

[BCGH06] Olivier Bournez, Manuel L. Campagnolo, Daniel S. Graça, and Emmanuel Hainry. The general purpose analog computer and computable analysis are two equivalent paradigms of analog computation. In *Proceedings of the Third International Conference on Theory and Applications of Models of Computation*, TAMC'06, pages 631–643, Berlin, Heidelberg, 2006. Springer-Verlag.

[BDG88] José Luis Balcazar, Jose Diaz, and Joaquim Gabarro. *Structural Complexity 1.* Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[BDG90] José Luis Balcázar, Joseph Díaz, and Joaquim Gabarró. *Structural Complexity 2.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[BGP14] Olivier Bournez, Daniel S. Graça, and Amaury Pouly. Continuous time models are equivalent to Turing machines. A characterization of P and NP with ordinary differential equations. April 2014. Preprint : `http://perso.ens-lyon.fr/yassine.hamoudi/wp-content/uploads/2014/08/pnpgpac.pdf`.

[BMS81] Alberto Bertoni, Giancarlo Mauri, and Nicoletta Sabadini. A characterization of the class of functions computable in polynomial time on Random Access Machines. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 168–176, New York, NY, USA, 1981. ACM. (errata : `http://perso.ens-lyon.fr/yassine.hamoudi/wp-content/uploads/2014/06/errata.pdf`).

[CG09] P. Collins and D. S. Graça. Effective computability of solutions of differential inclusions — the ten thousand monkeys approach. *Journal of Universal Computer Science*, 15(6) :1162–1185, 2009.

[GCB08] D. S. Graça, M. L. Campagnolo, and J. Buescu. Computability with polynomial differential equations. *Adv. Appl. Math.*, 40(3) :330–349, 2008.

[PS76] Vaughan R. Pratt and Larry J. Stockmeyer. A characterization of the power of Vector Machines. *J. Comput. Syst. Sci.*, 12(2) :198–221, April 1976.

Annexes

A Remarques générales sur le stage

Ce rapport a été produit suite à un stage de 6 semaines réalisé au sein du Laboratoire d'Informatique de l'Ecole Polytechnique (LIX) sous l'encadrement d'Olivier Bournez. Ce travail conclue l'année de L3 et donnera lieu à une présentation orale.

J'ai débuté mon stage en me familiarisant avec quelques notions de complexité, à l'aide notamment de [BDG88]. En parallèle, j'ai étudié le papier [BGP14], dont M. Bournez est l'un des auteurs, et qui a constitué la base de mes recherches. Ce travail bibliographique a été résumé dans les parties 2.1 à 2.3. L'objectif fut ensuite de produire une caractérisation de PSPACE par équations différentielles, comme c'est le cas pour les classes P et NP dans [BGP14]. L'utilisation d'oracles m'a permis d'obtenir un premier résultat exposé partie 2.4. Afin d'obtenir une autre caractérisation, peut-être plus élégante, j'ai cherché par la suite à simuler par équations différentielles une machine discrète reconnaissant PSPACE en temps polynomial. A ce titre, j'ai étudié dans un premier temps la thèse du calcul parallèle et les machines à vecteurs dans [BDG90] et [PS76]. J'ai découvert ensuite les RAM arithmétiques, qui convenait plus à mon travail, et qui m'ont permis de construire le modèle des automates arithmétiques, décrit partie 3.4. La simulation en elle-même de ces derniers a été entreprise, mais je n'ai pas pu mener ce travail à terme comme précisé partie 5.

Ces six semaines de stage furent très enrichissantes. Ce fut l'occasion pour moi de découvrir le monde de la recherche au quotidien, à travers notamment des échanges réguliers avec mon encadrant, Olivier Bournez, et son doctorant, Amaury Pouly. J'ai également beaucoup apprécié l'aspect original du sujet de stage, qui abordait des questions de complexité et de calculabilité du point de vue des équations différentielles, cherchant ainsi à mêler monde discret et monde continu.

Enfin, je tient à remercier Olivier Bournez pour son accueil et son investissement tout au long du stage. Ses nombreux conseils me seront d'une grande aide dans mes études et mon travail futurs. Je remercie également Amaury Pouly pour le temps qu'il m'a consacré.