Gymnase de Renens

Travail de maturité 2024

Conception et programmation d'un ordinateur 8 bits

Supervisé par Micha Hersch

Écrit par Lukan Morel

Renens, 31 février 2024

Table des matières

1 Introduction 2

2 Architecture de la machine 2
2.1 Introduction à l'architecture de CPU . . . 2
2.2 RAM . . . 3
2.3 Stack . . . 3
2.3.1 Cadre . . . 3
2.4 Registres . . . 4
2.5 Jeu d'instructions . . . 5
2.6 Exemple de programme . . . 6

3 Opérations sur entiers 7
3.1 Addition . . . 7
3.2 Soustraction . . . 7
3.3 Décalage de bits . . . 8
3.4 Multiplication . . . 8

4 Opérations sur nombres non entiers 9
4.1 Représentation à virgule flottante . . . 9
4.2 Addition et soustraction de floats . . . 10
4.3 Multiplication de floats . . . 10
4.4 Division de floats . . . 10
4.4.1 La méthode de Newton . . . 11
4.4.2 Approximation de la réciproque . . . 11
4.5 Calcul de fonctions trigonométriques . . . 13
4.5.1 Séries de Taylor . . . 13
4.5.2 Application des séries de Taylor pour le calcul de fonctions trigonométriques . . . 13

5 Calcul de décimales de pi 15

1 Introduction

Les résultats exacts, étant parfois difficiles voir impossibles à atteindre, nécessitent, pour s'en approcher, des approximations. Il s'agit donc de choisir, parmi un sous ensemble fini d'un ensemble qui ne l'est pas, la valeur qui correspond le mieux au résultat désiré. Le domaine mathématique de l'analyse numérique décrit les meilleurs manières de faire ce choix et plus récemment, décrit aussi comment déléguer cette décision aux machines. En effet, avec une importance moindre apportée à l'exactitude mais un intérêt grandissant pour la vitesse, les machines sont bien plus adaptées à cette tache que les humains. Dans ce travail, nous introduirons abstraitement la notion de processeur sans traiter des composants spécifiques à l'implémentation physique d'une telle machine. Ensuite nous montrerons comment, en partant d'opérations basiques et en se servant de l'abstraction comme échelon, nous pouvons atteindre n'importe quel sommet de complexité mathématique. Sans les résultats présentés dans ce travail, le monde moderne serait tout simplement impossible. L'entièreté de l'informatique, et donc par dépendance, de notre monde est construite sur les concepts fondamentaux que nous verrons ensemble ici. Notons que ce travail aura un accent particulier sur la compréhension et l'intuition plutôt que sur la rigueur. Laissons la rigueur aux artistes qui nous offrent ces résultats dont on profite si volontiers.
Ces sommets mathématiques étant, malgré leur élégance, très accessibles, un grand bagage de connaissances n'est pas nécessaire. Il s'agirait tout de même d'être familier avec quelques notions basiques d'analyse tel que la dérivation et les propriétés des fonctions trigonométriques et logarithmiques. Les autres résultats sont expliqués intuitivement avec plus ou moins de rigueur, en fonction de l'impact sur le niveau de compréhension. Les démarches complètes seront, bien évidemment, référencés dans la bibliographie.

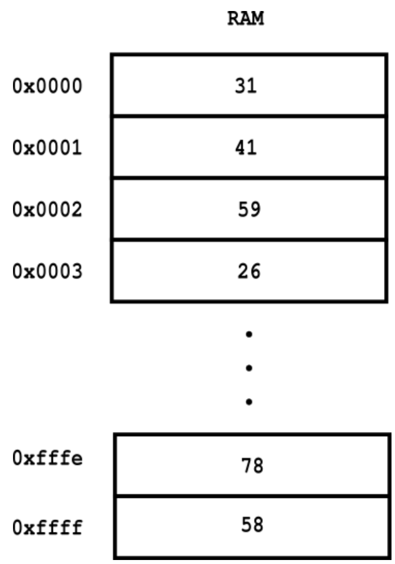
2 Architecture de la machine

2.1 Introduction à l'architecture de CPU

L'objectif de cette section est de concevoir une machine (qu'on appellera processeur), certes théorique mais tout à fait réaliste, qui sait effectuer un programme informatique. Un programme informatique est une suite d'instructions simples, stockée dans la mémoire externe, qui ensemble, permettent d'exécuter un algorithme bien plus complexe. L'ensemble de ces instructions est appelé le jeu d'instructions. De par sa taille et complexité, le jeu d'instructions définit les capacités du processeur: les ordinateurs d'aujourd'hui ont donc un jeu d'instructions bien plus grand et complexe que celui utilisé ici. Le jeu d'instructions comporte plusieurs types d'instructions que l'on peut classer en trois grandes familles: les instructions logiques et arithmétiques, les instructions de gestion de mémoire et les instructions de contrôle d'exécution. Les instructions logiques et arithmétiques permettent au processeur d'effectuer des opérations mathématiques. Il est important de noter que ces instructions n'opèrent pas directement sur les données stockées en mémoire externe. Au lieu de cela, les valeurs nécessaires sont transférées depuis la mémoire externe vers la mémoire interne du processeur (appelée les registres), sur laquelle on peut opérer. Puis, une fois les opérations terminées, les valeurs sont renvoyées dans la mémoire externe. Ceci est dû au fait que l'accès à la mémoire externe est bien plus lent que celui à la mémoire interne. Mais là où la mémoire interne gagne en vitesse, elle s'écrase lamentablement pour ce qui est de la taille. Ce compromis soulève alors un besoin pour des instructions dédiées aux transferts entre la mémoire externe et interne: les instructions de gestion de mémoire. Finalement, les instructions de contrôle d'exécution proposent une alternative à l'exécution linéaire des instructions. En effet, elles permettent de faire déplacer l'exécution vers une autre partie du programme. Ce déplacement est appelé un branchement. Un branchement devient particulièrement utile lorsqu'il est soumis à une condition. La valeur de vérité d'une proposition mathématique qui implique, ou non, un branchement permet au programme de prendre des décisions sur la suite de l'exécution en fonction du début. Ces branchements conditionnels sont un concept capital pour tout programme informatique. Armé d'un jeu d'instructions qui comporte au minimum ces trois types d'instructions et d'un processeur qui sait les exécuter, l'on a ce que celles et ceux qui l'ont conçu avant nous ont appelé, un CPU (Central Processing Unit). Ou unité centrale de calcul en français.
Les prochaines sous-sections auront pour but d'expliquer plus en détail les différents composants du processeur et de décrire le jeu d'instructions sur lequel nous allons travailler.

2.2 RAM

La RAM (Random Access Memory) ou mémoire externe, contient les informations nécessaires à l'exécution d'un programme (code du programme, variables, etc.). Les données sont séparées en blocs de un octet. Chaque octet est assigné une adresse de 16 bits, soit 2 octets, on a donc une mémoire totale de 64 KiB. ($2^{16} = 65536$)
Les adresses sont souvent représentées en hexadécimal et sont donc préfixées de "0x".



Lors de l'exécution d'un programme, le code se trouve au début de la mémoire, puis viennent les données numériques. Le stack se trouve à la fin et grandit en direction du début.

2.3 Stack

Le stack ou pile, est une structure de données, placée dans la RAM, qui opère sur le principe suivant: dernier rentré, premier sorti.
Nous pouvons l'imaginer comme une pile d'assiettes: l'on peut ajouter et récupérer des assiettes seulement depuis le haut de la pile. Ces deux opérations sont gérées nativement par les instructions `push`[1] et `pop` respectivement.
L'adresse du haut du stack est stockée, relativement au cadre actuel, dans le registre `sp`[2]. L'adresse absolue est alors donnée par `fp - sp`. Le stack sert principalement à stocker les données specifiques à l'appel d'une fonction.

2.3.1 Cadre

Un cadre est un espace de mémoire dans le stack réservé au données spécifiques à l'appel d'une fonction. Quand une fonction est appelée, un nouveau cadre, ainsi que l'addresse de retour, est placé sur le stack. Toutes les variables spécifiques ou locales seront stockées dans le cadre. Une fois la fonction terminée elles seront oubliées et le programme reprendra depuis l'addresse de retour.

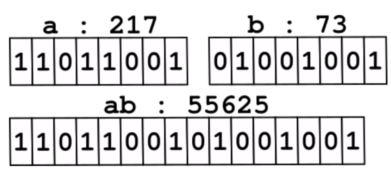
[1] Voir Instructions
[2] Voir Registres

2.4 Registres

Un registre est un emplacement de mémoire interne au processeur dont l'accès est très rapide. La plupart des instructions[1] opèrent directement sur les registres. Il y a 8 registres, de un octet chacun, à notre disposition, listés ci-dessous:

- Registre `a`, registre à usage général.
- Registre `b`, registre à usage général.
- Registre `c`, registre à usage général.
- Registre `d`, registre à usage général.
- Registre `hfp` (High Frame Pointer), stocke le premier octet de l'adresse du cadre actuel.
- Registre `lfp` (Low Frame Pointer), stocke le dernier octet de l'adresse du cadre actuel.
- Registre `sp` (Stack Pointer), stocke l'adresse du haut du stack relativement au cadre actuel.
- Registre `f` (Flag), stocke l'état des huit drapeaux:
  - `positive` (Positif), le résultat de l'opération est positif.
  - `carry` (Retenue), l'opération mène à une retenue.
  - `equal` (Egal), le registre est égal au registre ou à la valeur à laquelle il est comparé.
  - `zero` (Zéro), le résultat l'opération est nul.
  - `less` (Moins), le registre est inférieur au registre ou à la valeur à laquelle il est comparé.
  - `borrow` (Emprunt), la soustraction mène à un emprunt.
  - `negative` (Négatif), le résultat de l'opération est strictement négatif.
  - `overflow` (Débordement), le résultat de l'opération ne rentre pas dans un registre.

Les drapeaux `negative` et `overflow` sont utiles qu'en arithmétique signée, ou les valeurs sont représentées en complément à 2. Étant donné que ces drapeaux ne sont jamais utilisés dans le reste du travail, l'on peut ne pas les considérer. Les opérations; `add`, `sub`, `adc`, `sbb`, `iec`, `bsl` et `bsr` provoquent une mise à jour des drapeaux.
Les paires de registres (`a`, `b`) et (`c`, `d`) sont souvent utilisés ensemble et forment alors deux registres de 2 octets chacun nommés `ab` et `cd`, respectivement. La valeur de `ab` est donnée par $\texttt{a} \cdot 256 + \texttt{b}$ et pareillement pour `cd`.



[1] Voir Instructions

2.5 Jeu d'instructions

Le jeu d'instructions regroupe la totalité des opérations que la machine peut effectuer. En combinant ces instructions il est possible d'implémenter n'importe quel algorithme. Ci-dessous sont décrits les 16 instructions natives au processeur.

- `ldw` (Load Word), copie le contenu d'une adresse ou celle décrite par les registres `ab` ou `cd` dans un registre.
- `stw` (Store Word), copie le contenu d'un registre dans une adresse ou celle décrite par les registres `ab` ou `cd`.
- `mvw` (Move Word), copie une valeur immédiate ou le contenu d'un registre dans un autre registre.
- `add` (Add), additionne une valeur immédiate ou le contenu d'un registre à un autre registre et copie le résultat dans le premier registre.
- `sub` (Subtract), soustrait une valeur immédiate ou le contenu d'un registre à un autre registre et copie le résultat dans le premier registre.
- `adc` (Add with Carry), effectue l'opération `add` en ajoutant à l'addition la valeur du drapeau `carry`.
- `sbb` (Subtract with Borrow), effectue l'operation `sub` en ajoutant à la soustraction la valeur du drapeau `borrow`.
- `iec` (Increment/Decrement), incrémente ou décrémente de 1 le contenu d'un registre et stocke le résultat dans ce dernier.
- `cmp` (Compare), compare le contenu d'un registre avec le contenu d'un autre ou d'une valeur immédiate et met à jour les drapeaux appropriés.
- `jnz` (Jump if Not Zero), effectue un branchement vers une adresse ou celle décrite par le registre `ab` ou `cd` si le drapeau `zero` n'est pas activé.
- `push` (Push), copie le contenu d'un registre ou celui d'une adresse en haut de la pile et incrémente de 1 le contenu du registre `sp`.
- `pop` (Pop), copie le contenu du haut de la pile dans un registre ou une adresse et décrémente de 1 le contenu du registre `sp`.
- `bsl` (Bit Shift Left), décale vers la gauche les bits d'un registre ou des registres `ab` ou `cd` et copie le résultat dans ce(s) dernier(s).
- `bsr` (Bit Shift Right), décale vers la droite les bits d'un registre ou des registres `ab` ou `cd` et copie le résultat dans ce(s) dernier(s).
- `out` (Out), affiche le contenu d'un registre comme une valeur absolue ou comme une valeur en complément à 2.
- `halt` (Halt), halte l'éxecution du programme.

Un programme est constitué d'une séquence de ces instructions. Mais puisque la RAM qui contient le programme ne peut que stocker des nombres sous forme de bytes, chaque instruction doit être traduite en un nombre unique qui sera lui stocké. De cette manière, le processeur pourra comprendre les programmes qu'on lui donne. Cette traduction, qu'on appelle l'assemblage, ne sera pas expliquée ici. [1]

2.6 Exemple de programme

Mettons maintenant en œuvre ces instructions. Voyons un exemple de programme que notre machine peut effectuer. Il s'agit ici de calculer un certain nombre de termes de la suite de Fibonacci. Notons pour rappel que la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie comme tel:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

Voici alors le programme:

```
mvw a, 0 ; initialise le registre a avec 0
mvw b, 1 ; initialise le registre b avec 1

mvw d, 5 ; nombre de termes calculés

loop:
  out a ; affiche le contenu de a
  add a, b ; additionne a et b
  ; inverse les contenus de a et b
  mvw c, a ; déplace le contenu de a vers c
  mvw a, b ; déplace le contenu de b vers a
  mvw b, c ; déplace le contenu de c vers b
  dec d ; décrémente le nombre de termes restants
jnz loop ; relance la boucle sauf si d = 0
halt ; arrete le programme
```

Ce programme, une fois assemblé et exécuté par le processeur, affichera à la console: 0, 1, 1, 2, 3. Soit les 5 premiers termes de la suite de Fibonacci. Ici, la valeur de 5 est choisie arbitrairement. Le programme aurait pu calculer jusqu'à n'importe quel terme, tant que ce terme n'excède pas 255. Dans quel cas les résultats affichés seraient réduits modulo 256. Dans la prochaine section nous traiterons en détail plus spécifiquement des instructions arithmétiques et leurs impacts sur le registre des drapeaux.

L'idée derrière les ordinateurs numériques peut être expliquée en disant que ces machines sont destinées à effectuer toutes les opérations qui pourraient être effectuées par un ordinateur humain.

- Alan Turing

3 Opérations sur entiers

3.1 Addition

L'addition est une opération native au processeur, gérée par les instructions `add` et `adc`. L'instruction est utilisée comme-tel:

$$\texttt{add x, y}$$

avec `x` un registre et `y` un registre ou une valeur immédiate.
Alors le résultat de l'opération sera stocké dans le registre `x`. Il est important de noter que puisque le registre ne peut que stocker des valeurs allant jusqu'à 255, le résultat de l'opération sera modulo 256. Soit,

$$\texttt{add x, y} \implies \mathtt{x} \leftarrow \mathtt{x} + \mathtt{y} \mod 256$$

Si le résultat de `x + y` dépasse 255, alors le drapeau `carry` sera activé. On peut donc le considérer comme le neuvième bit du résultat. Son activation est décrite par,

$$\texttt{add x, y} \implies \texttt{carry} = \begin{cases} 1, & \mathtt{x} + \mathtt{y} > 255 \\ 0, & \text{sinon.} \end{cases}$$

Si le résultat de `x + y` est nul, alors le drapeau `zero` sera activé. Son activation est décrite par,

$$\texttt{add x, y} \implies \texttt{zero} = \begin{cases} 1, & \mathtt{x} + \mathtt{y} \mod 256 = 0 \\ 0, & \text{sinon.} \end{cases}$$

L'instruction `adc` fonctionne pareillement à `add` mais elle additionne en plus la valeur du drapeau `carry`.

3.2 Soustraction

La soustraction est une opération native au processeur, gérée par les instructions `sub` et `sbb`. L'instruction est utilisée comme-tel:

$$\texttt{sub x, y}$$

avec `x` un registre et `y` un registre ou une valeur immédiate.
Alors le résultat de l'opération sera stocké dans le registre `x`. Comme pour l'addition le résultat est modulo 256, ce qui permet de gèrer les résultats négatifs. Soit,

$$\texttt{sub x, y} \implies \mathtt{x} \leftarrow \mathtt{x} - \mathtt{y} \mod 256$$

Pour indiquer un résultat négatif, le drapeau `borrow` est activé. Son activation est décrite par,

$$\texttt{sub x, y} \implies \texttt{borrow} = \begin{cases} 1, & \mathtt{x} - \mathtt{y} < 0 \\ 0, & \text{sinon.} \end{cases}$$

Si le résultat de `x - y` est nul, alors le drapeau `zero` sera activé. Son activation est décrite par,

$$\texttt{sub x, y} \implies \texttt{zero} = \begin{cases} 1, & \mathtt{x} - \mathtt{y} \mod 256 = 0 \\ 0, & \text{sinon.} \end{cases}$$

L'instruction `sbb` fonctionne pareillement à `sub` mais elle soustrait en plus la valeur du drapeau `borrow`.

3.3 Décalage de bits

Le décalage de bits est une opération native au processeur, gérée par les instructions `bsl` et `bsr`. L'instruction est utilisée comme-tel:

$$\texttt{bsl x}$$

avec x un registre. Alors le résultat sera stocké dans le registre `x`. Le décalage de bits consiste a décaler les bits d'une valeur vers la gauche ou vers la droite. L'opération est notée `x << 1` pour un décalage vers la gauche ou `x >> 1` pour un décalage vers la droite. L'on peut considérer ces opérations comme une multiplication par deux et une division à partie entière par deux respectivement. Soit,

$$\begin{aligned}\texttt{bsl x} &\implies \texttt{x} \leftarrow 2\texttt{x} \mod 256\\ \texttt{bsr x} &\implies \texttt{x} \leftarrow \left\lfloor \frac{\texttt{x}}{2} \right\rfloor\end{aligned}$$

Si, après un décalage, un bit a été déplacé hors du registre, que ce soit vers la gauche ou la droite, le drapeau `carry` sera activé comme-tel:

$$\texttt{bsl x} \implies \texttt{carry} = \begin{cases} 1, & \texttt{x << 1} > 255 \\ 0, & \text{sinon.} \end{cases}$$

$$\texttt{bsr x} \implies \texttt{carry} = \begin{cases} 1, & \texttt{x >> 1} < \frac{\texttt{x}}{2} \\ 0, & \text{sinon.} \end{cases}$$

Si le résultat d'un décalage est nul, alors le drapeau `zero` sera activé. Son activation est décrite par,

$$\texttt{bsl x} \implies \texttt{zero} = \begin{cases} 1, & \texttt{x << 1} = 0 \\ 0, & \text{sinon.} \end{cases}$$

$$\texttt{bsr x} \implies \texttt{zero} = \begin{cases} 1, & \texttt{x >> 1} = 0 \\ 0, & \text{sinon.} \end{cases}$$

Il est également possible de décaler deux registres ensemble comme si ils n'étaient qu'un avec `bsl ab`, `bsl cd` et `bsr ab`, `bsr cd`.

3.4 Multiplication

Dans le reste du travail, la multiplication d'entiers sera occasionnellement utile. On distingue alors deux cas de figure. Celui ou l'on veut multiplier une variable par une constante et celui ou l'on veut multiplier une variable par une autre variable. Pour le premier cas on alors l'opération $x \cdot c$, ou c est une constante. En réécrivant c comme une somme de puissances de 2 on a

$$x \cdot (a_0 2^0 + a_1 2^1 + \cdots + a_7 2^7), \text{ avec } a_k = 1 \text{ ou } 0.$$

On distribue pour obtenir

$$x \cdot a_0 2^0 + x \cdot a_1 2^1 + \cdots + x \cdot a_7 2^7.$$

Et puisque $x \cdot 2^n$ est égal à n décalages de x vers la gauche, le problème est résolu. Par exemple, multiplier x par 10 revient à évaluer x<<1 + x<<3.
Le deuxième cas est plus complexe. La multiplication par addition répétée est souvent découragée car elle devient beaucoup moins efficace que d'autres méthodes plus sophistiquées quand les facteurs deviennent de plus en plus grand. Mais étant donné que nous travaillons sur des valeurs plutôt petites, la simplicité offerte par cette méthode naïve en vaut la peine. C'est alors par addition répétée que nous procéderons.

4 Opérations sur nombres non entiers

Afin de répondre à un plus grand nombre de problèmes mathématiques il sera utile de travailler non pas que sur des entiers mais également sur les rationaux qui les entourent. Il s'agit donc de définir une représentation et des opérations élémentaires pour ces valeurs fractionnelles.

4.1 Représentation à virgule flottante

Le standard utilisé universellement pour représenter des valeurs fractionnelles dans les ordinateurs est celui de la virgule flottante (ou floating point en anglais). Les valeurs de ce type sont appelées des floats. Un float est composé de trois parties:

- un signe, $s \in \{-1, 1\}$;
- un exposant, E;
- une mantisse, M.

Ces trois parties sont telles que la valeur du float est donnée par $s \cdot M \cdot 2^E$.
Ce système s'apparente à être l'équivalent de la notation scientifique pour le binaire. Il reste maintenant la question de comment encoder ces trois parties dans un espace fini et compact. Nous allons choisir comme taille 16 bits. L'encodage du signe s est intuitif, nous pouvons y dédier le premier bit (0 si s = 1, 1 si s = -1). La taille de l'exposant déterminera l'ordre de grandeur minimal et maximal de nos floats. Si l'on attribue 5 bits à l'exposant, ce dernier peut avoir $2^5 = 32$ valeurs différentes. Étant donné qu'il nous faut aussi des exposants négatifs nous pouvons ajouter un biais à l'exposant. C'est-à-dire que l'exposant $E = E_s - 15$ où E_s est l'entier véritablement représente par les 5 bits dédiés. Les valeurs sont donc bornées par 2^{-15} et 2^{16}. Il nous reste alors 10 bits pour la mantisse. Notons que, comme en notation scientifique, la mantisse M doit pouvoir représenter des valeurs non entières dans l'intervalle $[1; 2[$. Notons ensuite que toutes les valeurs incluses dans cet intervalle ont comme partie entière 1. Puisque la partie entière est constante, il n'est pas nécessaire de la stocker. Soit, la mantisse $M = 1 + 2^{-10}M_s)$ où M_s est l'entier véritablement stocké par les 10 bits dédiés. Finalement, la valeur d'un float est donnée par

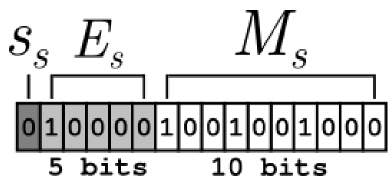
$$(-1)^{s_s}\left(1 + \frac{M_s}{2^{10}}\right) \cdot 2^{E_s - 15}, \text{ ou } s_s \text{ est la valeur du premier bit.}$$

Cette formule n'est pas générale, dans certain cas spéciaux le float doit être interprété différemment. Ceci est nécessaire pour représenter des valeurs comme 0 ou $\pm\infty$. Plus généralement la valeur d'un float est donnée par

$$\begin{cases} (-1)^{s_s}(1 + \frac{M_s}{2^{10}}) \cdot 2^{E_s - 15}, & E_s \notin \{0, 31\} \\ (-1)^{s_s}\frac{M_s}{2^{10}} \cdot 2^{E_s - 15}, & E_s = 0 \text{ et } M_s \neq 0 \\ 0, & s_s = 0,\ E_s = 0 \text{ et } M_s = 0 \\ -0, & s_s = 1,\ E_s = 0 \text{ et } M_s = 0 \\ +\infty, & s_s = 0,\ E_s = 31 \text{ et } M_s = 0 \\ -\infty, & s_s = 1,\ E_s = 31 \text{ et } M_s = 0 \\ \text{NaN}, & E_s = 31 \text{ et } M_s \neq 0 \end{cases}$$

NaN (Not a Number), désigne une valeur indéterminée comme par exemple le résultat de $\frac{0}{0}$.

En mémoire un float est disposé comme tel



$$(-1)^{s_s}\left(1+\frac{M_s}{2^{10}}\right)\cdot 2^{E_s-15}=(-1)^0\left(1+\frac{584}{2^{10}}\right)\cdot 2^{16-15}=1.57\cdot 2=3.14$$

Il s'agit maintenant de définir des procédures pour opérer sur ces floats.

4.2 Addition et soustraction de floats

Pour additionner deux floats α et β en un float Γ, il faut premièrement extraire les signes, exposants et mantisses des floats α et β. Ensuite, en établissant la relation suivante

$$s_\Gamma\cdot M_\Gamma\cdot 2^{E_\Gamma}=s_\alpha\cdot M_\alpha\cdot 2^{E_\alpha}+s_\beta\cdot M_\beta\cdot 2^{E_\beta}$$

on remarque que si $E_\alpha=E_\beta$, le problème devient trivial. Il suffit d'abord de convertir les mantisses de α et β en complément à 2 en fonction du signe, puis de les additionner. Si le résultat de l'addition de mantisses est négatif il faudra reconvertir le résultat du complément à 2 et $s_\Gamma=-1$. Finalement on décale les bits du résultat de sorte à ce que la partie entière soie 1 et on change l'exposant en conséquence. Le résultat et l'exposant normalisés nous donnent M_Γ et E_Γ respectivement. Le cas $E_\alpha\neq E_\beta$ ajoute une étape avant l'addition des mantisses. Il faut d'abord décaler les bits de la mantisse du plus petit nombre $|E_\alpha-E_\beta|$ fois vers la droite.

La soustraction de floats suit le même processus que l'addition, à l'exception que l'addition de mantisses est substituée, sans surprise, par une soustraction. Dans le cas ou le résultat dépasse les bornes de notre représentation, l'on peut simplement retourner plus ou moins l'infini.

4.3 Multiplication de floats

Pour multiplier deux floats α et β en un float Γ, il faut premièrement extraire les signes, exposants et mantisses des floats α et β. Ensuite, en établissant la relation suivante

$$\begin{aligned}s_\Gamma\cdot M_\Gamma\cdot 2^{E_\Gamma}&=s_\alpha\cdot M_\alpha\cdot 2^{E_\alpha}\ \cdot\ s_\beta\cdot M_\beta\cdot 2^{E_\beta}\\&=s_\alpha s_\beta\cdot M_\alpha M_\beta\cdot 2^{E_\alpha+E_\beta}\end{aligned}$$

on remarque qu'il suffit de multiplier les signes, les mantisses et d'additionner les exposants, puis de normaliser le résultat pour obtenir Γ. Dans le cas ou le résultat dépasse les bornes de notre représentation, l'on peut simplement retourner plus ou moins l'infini.

4.4 Division de floats

Pour diviser deux floats nous allons multiplier l'un par l'inverse de l'autre. Nous détaillerons ci-dessous une manière très efficace de trouver une approximation de la réciproque d'un float. Cette approximation pourra ensuite être améliorée avec quelques itérations de la méthode de Newton.

4.4.1 La méthode de Newton

La méthode de Newton est un algorithme efficace pour trouver numériquement les zeros d'une fonction. Pour une fonction à variable réelle f, un x_0 proche du zéro à trouver et un f dérivable et de dérivée non nulle en x_0, généralement, une meilleur estimation du zéro est trouvable par:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Cette méthode peut ensuite être itérée pour obtenir un résultat aussi proche que désiré:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

4.4.2 Approximation de la réciproque

Pour calculer la réciproque d'un float, notons d'abord qu'un float α est un nombre de 16 bits que l'on interprète comme un float. Dans quel cas, la valeur de α, qu'on appelle α_f est donnée par

$$\alpha_f = \left(1 + \frac{M_{s,\alpha}}{2^{10}}\right) \cdot 2^{E_{s,\alpha}-15}.$$ [1]

Mais si l'on interprète α comme un entier, sa valeur α_i (appelée representation entière de α) est donnée par

$$\alpha_i = 2^{10} E_{s,\alpha} + M_{s,\alpha}.$$

En sachant ceci, prenons maintenant le logarithme base 2 de α_f

$$\begin{aligned}\log_2(\alpha_f) &= \log_2\left(\left(1 + \frac{M_{s,\alpha}}{2^{10}}\right) \cdot 2^{E_{s,\alpha}-15}\right) \\ &= \log_2\left(1 + \frac{M_{s,\alpha}}{2^{10}}\right) + E_{s,\alpha} - 15\end{aligned}$$

Remplaçons maintenant le premier terme par une approximation pour $\log_2(1+x)$, soit $x + \mu$ où μ est tel qu'il minimise $\max_{0 \le x \le 1} |\log_2(1+x) - (x+\mu)|$

$$\begin{aligned} &\approx \frac{M_{s,\alpha}}{2^{10}} + \mu + E_{s,\alpha} - 15 \\ &\approx \frac{1}{2^{10}}(2^{10} E_{s,\alpha} + M_{s,\alpha}) + \mu - 15 \\ \log_2(\alpha_f) &\approx \frac{1}{2^{10}}(\alpha_i) + \mu - 15.\end{aligned}$$

Cette relation nous dit que le logarithme base 2 d'un float est directement lié à son entier associé, ce qui s'avérera être très utile par la suite. Néanmoins il s'agirait de clarifier l'approximation $\log_2(1+x) \approx x + \mu$ et de déterminer comment trouver une valeur numérique pour μ. On cherche donc un polynôme $p(x) = \lambda x + \mu$, tel qu'il soit le polynôme de degré 1 qui approxime la fonction $f(x) = \log_2(1+x)$ le mieux possible sur l'intervalle $[0;1]$. On choisit cette intervalle puisque dans notre cas, x ne varie qu'entre 0 et 1.

[1] Le signe n'est pas pris en compte, on admet que le float est positif. Dans le cas contraire, il suffit de multiplier par -1 le résultat obtenu.

Mathématiquement on dit que les coefficients λ et μ sont tels qu'ils minimisent $\max_{0\leq x\leq 1}|f(x)-p(x)|$. Ce polynôme est alors appelé le polynôme mini-max d'ordre 1 de $f(x)$. Introduisons maintenant un théorème qui sera nécessaire pour procéder.
Le théorème d'équioscillation de Chebyshev nous dit:
Pour une fonction f continue de $[a;b]$ vers $\mathbb{R}$, un polynôme de degré $\leq n$ est un polynôme mini-max d'ordre n de f si et seulement si il y a $n{+}2$ points $a \leq x_0 < x_1 < \cdots < x_{n+1} \leq b$ tels que $f(x_k)-p(x_k) = \sigma(-1)^k \max_{0\leq x\leq 1}|f(x)-p(x)|$ avec $\sigma = 1$ ou -1.
Soit il y a $n+2$ points ou l'erreur maximale est atteinte et l'erreur alterne en signe. Nous ne détaillerons pas la preuve ici, car elle dépasse le cadre du travail. Dans notre cas, puisqu'on cherche un polynôme de degré 1, il y a 3 points $0 \leq x_0 < x_1 < x_2 \leq 1$ où l'erreur maximale est atteinte. Essayons de trouver ces points pour procéder. On cherche donc les valeurs maximales et minimales de $E(x) = f(x) - p(x)$. Trouver les extremums d'une fonction revient à trouver les zéros de la dérivée de cette fonction. Ici,

$$E'(x) = \frac{1}{\ln 2 \cdot (1+x)} - \lambda$$

Si $E'(x) = 0$, alors

$$x = \frac{1}{\lambda \cdot \ln 2} - 1$$

Puisque $E'(x)$ n'a qu'un seul zéro, les autres extremums se trouvent nécessairement aux bornes de l'intervalle soit,

$$x_0 = 0,\ x_1 = \frac{1}{\lambda \cdot \ln 2} - 1,\ x_2 = 1.$$

Chebyshev nous dit alors que $E(x_0) = -E(x_1)$ et que $E(x_1) = -E(x_2)$. Ces égalités nous donnent le système suivant:

$$\begin{cases} E(x_0) = -E(x_1) \\ E(x_2) = -E(x_1) \end{cases} \implies \begin{cases} \mu = \frac{1}{2}(\log_2(\frac{1}{\lambda\cdot\ln 2}) - \frac{1}{\ln 2} + \lambda) \\ \mu = \frac{1}{2}(\log_2(\frac{1}{\lambda\cdot\ln 2}) - \frac{1}{\ln 2} + 1) \end{cases} \implies \lambda = 1.$$

En connaissant λ, on peut aisément calculer μ:

$$\begin{aligned} \mu &= \frac{1}{2}\Big(\log_2\Big(\frac{1}{\ln 2}\Big) - \frac{1}{\ln 2} + 1\Big) \\ &= \frac{\ln 2 - \ln(\ln 2) - 1}{2\ln 2} \\ &\approx 0.04304... \end{aligned}$$

Maintenant que nous avons trouvé une valeur pour μ procédons avec le calcul de l'inverse d'un float.
Soit un float α et un float Γ avec une valeur $\Gamma_f = \alpha_f^{-1}$.
En prenant le logarithme base 2 des deux cotés on obtient

$$\log_2(\Gamma_f) = -\log_2(\alpha_f)$$

En remplaçant $\log_2(\Gamma_i)$ par l'approximation établie plus haut,

$$\begin{aligned} \frac{1}{2^{10}}(\Gamma_i) + \mu - 15 &\approx -\frac{1}{2^{10}}(\alpha_i) - \mu + 15 \\ \frac{1}{2^{10}}(\Gamma_i) &\approx -\frac{1}{2^{10}}(\alpha_i) - 2\mu + 30 \\ \Gamma_i &\approx -\alpha_i - 2048\mu + 30720. \end{aligned}$$

Finalement, en remplaçant $-2048\mu + 30720$ par l'entier qui s'en rapproche le plus, $m = 30632$, on obtient

$$\Gamma_i \approx -\alpha_i + m.$$

Cette relation montre que l'on peut obtenir une approximation pour l'inverse d'un float en ne faisant que des opérations sur la représentation entière d'un float. Pour ensuite ameliorer l'approximation avec la méthode de Newton on a

$$\begin{cases}\Gamma_0 = -\alpha_i + m \\ \Gamma_{n+1} = \Gamma_n - \frac{f(\Gamma_n)}{f'(\Gamma_n)},\ f(x) = \frac{1}{x} - \alpha_f\end{cases} \implies \begin{cases}\Gamma_0 = -\alpha_i + m \\ \Gamma_{n+1} = \Gamma_n - \frac{\frac{1}{\Gamma} - \alpha_f}{\frac{1}{\Gamma^2}}\end{cases} \implies \begin{cases}\Gamma_0 = -\alpha_i + m \\ \Gamma_{n+1} = \Gamma_n(2 - \alpha_f \Gamma_n)\end{cases}$$

Voici alors une méthode complète pour calculer la réciproque d'un float et, muni de l'algorithme de multiplication décrit dans la dernière section, nous pouvons calculer la division de deux floats arbitraires.

4.5 Calcul de fonctions trigonométriques

Les méthodes d'approximation de fonctions trigonométriques sont variées, celle expliquée ci-dessous n'a pas tant été choisie pour son efficacité que pour son élégance et simplicité. Elle repose sur cette observation: Remarquons que pour l'instant les seules fonctions que notre machine peut évaluer sont celles uniquement composées des 4 opérations élémentaires, dont des polynômes. Pour cette raison l'évaluation des fonctions trigonométriques parait impossible. Il sera alors utile de définir un outil pour convertir les fonctions désirées en polynômes, qui sont eux évaluables.

4.5.1 Séries de Taylor

Les séries de Taylor sont un outil puissant pour approximer les fonctions avec des polynômes. Admettons que nous avons une fonction $f(x)$ et que nous voulons un polynôme $P_n(x)$ qui approxime $f(x)$ autour de a. C'est a dire que plus x s'éloigne de a, plus $P_n(x)$ s'éloigne de $f(x)$. Soit,

$$P_n(x) = c_0 + c_1(x-a) + c_2(x-a)^2 + \cdots + c_n(x-a)^n.$$

Les coefficients $c_0, \cdots, c_n$ restent alors à déterminer. Il parait intuitif que en $P_n(a)$ devrait être égal à $f(a)$. Puisque $P_n(a) = c_0$, on en déduit $c_0 = f(a)$. Ensuite, en continuant avec intuition, on décide que $P_n'(a) = f'(a)$. Et plus géneralement $P_n^{(k)}(a) = f^{(k)}(a)$ pour tout $k \leq n$, ou $f^{(k)}(x)$ désigne la k-ième dérivée de $f(x)$. En dérivant k fois $P_n(x)$ on obtient

$$P_n^{(k)}(x) = c_k k! + c_{k+1} k!(x-a)^{k+1} + \cdots + c_n(x-a)^n.$$

Alors $P_n^{(k)}(a) = c_k k!$, soit $c_k = \frac{f^{(k)}(a)}{k!}$.
Donc

$$P_n(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n.$$

Plus n est grand, plus l'approximation est bonne. On alors

$$f(x) = \sum_{k=1}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

La série associée ici à la fonction f est appelée la série de Taylor de f autour de a. Montrer qu'une fonction est égale à sa série de Taylor est un exercice plus compliqué et ne sera donc pas détaillé ici. Heureusement la grande partie des fonctions utilisées couramment le sont.

4.5.2 Application des séries de Taylor pour le calcul de fonctions trigonométriques

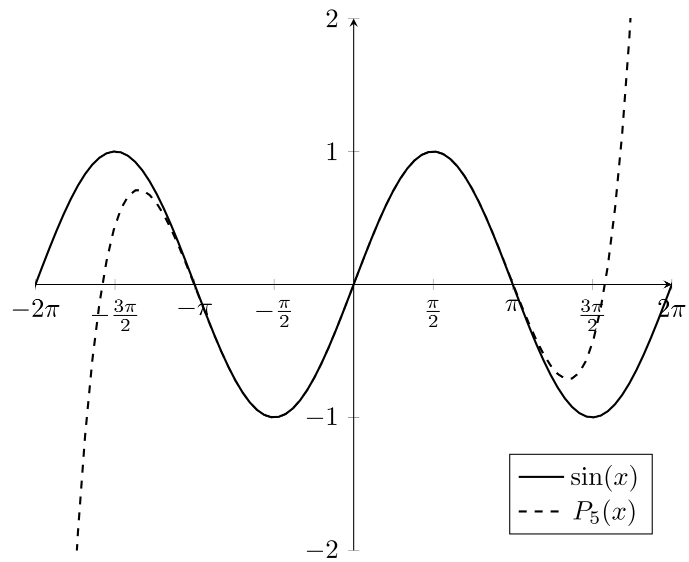
Maintenant que nous avons un outil pour convertir des fonctions en polynômes, essayons de l'appliquer sur les fonctions que nous voulons évaluer. Par exemple, en prenant $f(x) = \sin(x)$ et $a = 0$ on obtient la série de Taylor de $\sin(x)$ autour de 0,

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots.$$

Similairement, on peut obtenir la série de Taylor de $\cos(x)$ autour de 0,

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots.$$

En tronquant les séries ci-dessus à un nombre approprié de termes on obtient une expression qui est facilement évaluable par notre machine. Vérifions maintenant graphiquement la précision de ces approximations.



Comparons ici la fonction $\sin(x)$ et son approximation de Taylor tronquée à 5 termes, soit $x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$. On s'arrête à 5 termes car le prochain impliquerait une multiplication par $\frac{1}{11!} \approx 2.5 \cdot 10^{-8}$ ce qui est inférieur a la valeur minimale représentable par nos floats ($\approx 6 \cdot 10^{-8}$). Intuitivement, il semblerait qu'il nous faut juste une approximation sur le segment $[0; 2\pi]$ et ensuite la périodicité de la fonction permettrait une approximation sur tout les réels. Pourtant, la meilleure approximation possible avec nos limitations physiques commence déjà à se dégrader autour de $x = \pi$. Pour remédier à ce problème, remarquons qu'on peut séparer la fonction en segments de longueur π qui sont tous des symétries du premier segment $[-\frac{\pi}{2}; \frac{\pi}{2}]$ (sur lequel l'approximation semble plutôt rapprochée de la valeur exacte). C'est à dire que le premier segment encode toute l'information nécessaire pour représenter le reste de la fonction. Il faut alors un algorithme qui peut rabattre tout nombre sur son équivalent dans le segment $[-\frac{\pi}{2}; \frac{\pi}{2}]$. Soit, $x \to \theta$ avec $x \in \mathbb{R}$, $\sin(x) = \sin(\theta)$ et $\theta \in [-\frac{\pi}{2}; \frac{\pi}{2}]$. Ci-dessous est détaillé le pseudocode d'un tel algorithme:

$\theta \leftarrow x \mathbin{\%} 2\pi$
if $x > \pi$ **then**
    $\theta \leftarrow \pi - \theta$
end if

if $|\theta| > \frac{\pi}{2}$ **then**
    if $\theta < 0$ **then**
        $\theta \leftarrow \theta - \pi$
    else if $\theta \geq 0$ **then**
        $\theta \leftarrow \pi - \theta$
    end if
end if

Ici, l'opération %, présente à la première ligne de l'algorithme, dénote l'opération tel que la relation $x \% y = z$ soit vraie si ces 3 conditions sont réunies: $x = z + ky$, $k \in \mathbb{Z}$ et $0 \leq z < y$. Cette opération est facilement calculable par notre machine en effectuant $z = x - y \cdot \mathrm{trunc}(\frac{x}{y})$. Où $\mathrm{trunc}(x)$ est la fonction qui associe à un réel x, sa partie entière. Ce qui est plutôt simple à faire avec un float. Une fois θ trouvé, il suffit d'évaluer la série de Taylor en θ. Pour approximer $\cos(x)$ on utilise une méthode similaire, mais puisque $\cos(x)$ est positif sur le segment $[-\frac{\pi}{2}; \frac{\pi}{2}]$, il n'existe pas un θ compris dans ce segment tel que $\cos(x) = cos(\theta)$, $\forall x \in \mathbb{R}$. On peut en revanche trouver θ, tel que $|\cos(x)| = cos(\theta)$, et le signe du résultat final, qu'on applique au résultat obtenu après l'évaluation de la série.

5 Calcul de décimales de pi

En utilisant la répresentation float, la meilleur approximation possible de π est égale à 3.140625. Alors, essayer de calculer un maximum de décimales de π sous la forme d'un float est un exercice un peu décevant. Heureusement il existe des algorithmes qui ne travaillent que sur des entiers, et qui produisent, iterativement, des décimales de π. L'algorithme utilisé ici, découvert par Rabinowitz et Wagon en 1995 [2], est lui même basé sur un autre algorithme pour calculer les décimales de e, découvert par Sale en 1968 [3] que nous allons détailler en premier afin de mieux comprendre celui pour π. Cet algorithme se base sur la somme infinie ci-dessous.

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} \cdots$$

Cette série peut être notamment obtenue en évaluant la série de Taylor de e^x autour de 0, en $x = 1$. On peut ensuite réécrire cette série sous la forme

$$e = 1 + 1\Big(1 + \frac{1}{2}\Big(1 + \frac{1}{3}\Big(1 + \frac{1}{4}\Big(1 + \frac{1}{5}\Big(1 + \cdots\Big)\Big)\Big)\Big)\Big).$$

En tronquant, au besoin, le nombre de termes on obtient une approximation pour e. Prenons par exemple cette approximation:

$$\begin{aligned} e &\approx 1 + 1\Big(1 + \frac{1}{2}\Big(1 + \frac{1}{3}\Big(1 + \frac{1}{4}\Big(1 + \frac{1}{5}\Big(1\Big)\Big)\Big)\Big)\Big) \\ &\approx 2 + \frac{1}{2}\Big(1 + \frac{1}{3}\Big(1 + \frac{1}{4}\Big(1 + \frac{1}{5}\Big(1\Big)\Big)\Big)\Big) \end{aligned}$$

Remarquons ensuite que la partie fractionnelle de l'approximation, sous cette forme, est

$$\frac{1}{2}\Big(1 + \frac{1}{3}\Big(1 + \frac{1}{4}\Big(1 + \frac{1}{5}\Big(1\Big)\Big)\Big)\Big)$$

et que donc la partie entière est 2. Pour trouver la prochaine décimale, il suffit de multiplier la partie fractionnelle par 10, et de prendre sa partie entière. On a alors

$$\begin{aligned} &\frac{1}{2}\Big(10 + \frac{1}{3}\Big(10 + \frac{1}{4}\Big(10 + \frac{1}{5}\Big(10\Big)\Big)\Big)\Big) \\ &= 7 + \frac{1}{2}\Big(0 + \frac{1}{3}\Big(1 + \frac{1}{4}\Big(0 + \frac{1}{5}\Big(0\Big)\Big)\Big)\Big). \end{aligned}$$

La partie entière est donc de 7. En répétant l'opération sur la partie fractionnelle, l'on peut obtenir la prochaine décimale et ainsi de suite. Cet algorithme est très simple et n'opère que sur des nombres entiers. Essayons maintenant d'appliquer une méthode similaire pour les décimales de π. Premièrement il nous faut une représentation de π propice a cet algorithme. Celle que nous allons utiliser découle du produit de Wallis pour π.[1]

$$\frac{\pi}{2} = \sum_{k=1}^{\infty} \frac{k!}{(2k+1)!!}$$

[1] La dérivation, dépassant le cadre de ce travail, n'est pas détaillée ici.

ou $(2k+1)!!$ dénote le produit des entiers impairs de 1 a $2k+1$, $(2k+1)!! = 1 \cdot 3 \cdot 5 \cdots (2k+1)$. On en déduit alors une représentation propice a l'algorithme.

$$\begin{aligned}\frac{\pi}{2} &= 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \frac{1 \cdot 2 \cdot 3 \cdot 4}{3 \cdot 5 \cdot 7 \cdot 9} + \cdots \\ &= 1 + \frac{1}{3}\Big(1 + \frac{2}{5}\Big(1 + \frac{3}{7}\Big(1 + \frac{4}{9}\Big(1 + \cdots\Big)\Big)\Big)\Big) \\ \pi &= 2 + \frac{1}{3}\Big(2 + \frac{2}{5}\Big(2 + \frac{3}{7}\Big(2 + \frac{4}{9}\Big(2 + \cdots\Big)\Big)\Big)\Big).\end{aligned}$$

Contrairement à l'algorithme pour e, la partie entière de π n'est pas simplement donnée par le premier terme, car le deuxième terme n'est pas forcement inférieur à 1. Pour cette raison, quand une décimale est calculée il faut la garder en mémoire pour vérifier que la prochaine décimale n'est pas un 10. Dans quel cas on ajoute 1 à la décimale précédemment stockée. Opérons maintenant de la même manière que pour calculer les décimales de e:

$$\pi \approx 2 + \frac{1}{3}\Big(2 + \frac{2}{5}\Big(2 + \frac{3}{7}\Big(2 + \frac{4}{9}\Big(2\Big)\Big)\Big)\Big)$$

On stocke donc le 2 et l'on garde que le deuxième terme qu'on multiplie par 10.

$$\frac{1}{3}\Big(20 + \frac{2}{5}\Big(20 + \frac{3}{7}\Big(20 + \frac{4}{9}\Big(20\Big)\Big)\Big)\Big)$$

$$= 10 + \frac{1}{3}\Big(2 + \frac{2}{5}\Big(2 + \frac{3}{7}\Big(0 + \frac{4}{9}\Big(2\Big)\Big)\Big)\Big)$$

Puisque la prochaine décimale est un 10 on en déduit donc que la première décimale (ou chiffre dans ce cas) de π est un 3. En continuant de la même manière l'on peut aisément calculer le reste des décimales offertes par cette approximation. Cette explication omet, par volonté de simplifier, certains aspects. Mais l'idée générale reste la même. Pour programmer cette algorithme on représente l'approximation comme une liste de 2 sur laquelle il est facile de manipuler chaque entrée en fonction de sa position dans la liste. Il parait intuitif que plus la liste est grande, meilleure l'approximation sera et donc plus on aura de décimales correctes. Puisque dans notre cas l'espace de mémoire est une contrainte plutôt importante il sera important d'écrire un programme le plus court possible, de sorte à pouvoir utiliser le reste de la mémoire pour une liste de taille maximale. Dans notre cas il a été possible d'écrire un programme assez court pour que la liste aie 32'485 entrées de 16 bits. Ceci a résulté en une approximation de pi correcte jusqu'à 9723 décimales. Notons que ceci n'a absolument aucune utilité pratique, puisque seulement 38 décimales suffisent pour calculer le périmètre d'un disque de la taille de l'univers observable avec une précision de l'ordre du rayon d'un atome d'hydrogène. [4]

References

[1] David Salomon. *Assemblers and loaders*. Ellis Horwood, 1992.

[2] Stanley Rabinowitz and Stan Wagon. "A spigot algorithm for the digits of π". In: *The American mathematical monthly* 102.3 (1995), pp. 195–203.

[3] A. H. J. Sale. "The Calculation of e to Many Significant Digits". In: *The Computer Journal* 11.2 (1968), pp. 229–230.

[4] Kit Yates. *Why life is more interesting with extra pi*. Mar. 2024. URL: `https://www.bbc.com/future/article/20240313-pi-day-the-number-that-can-help-unravel-the-universe`. (accessed: 12.10.2024).