Project Wetenschappelijk Rekenen

Sys, Jonas

June 17, 2022

Contents

1 Nulpunten van Veeltermen

1.1 De companionmatrix

1.1.1 Stelling

$\forall p(x) : p(x)$ is een monische veelterm $\wedge gr(p) = n \geq 1$
$\implies \forall x_0 \in eigVl(C_p) : p(x_0) = 0$

1.1.2 Bewijs

Inductie:

Inductiebegin: $n = 1$

$$\begin{aligned} p_1(x) &= x + c_0 \implies \\ C_{p_1} &= [-c_0] \implies \\ \rho(C_{p_1}) &= det(C_{p_1} - \lambda * I) \\ &= -c_0 - \lambda \\ &= (-1)^1 p_1 \end{aligned}$$

Inductiestap: geldig voor graad $n \implies$ geldig voor graad $n+1$
Neem $p_{n+1}(x) = x^{n+1} + \sum_{i=0}^{n} (c_i * x^i)$ en $p_n(x) = x^n + \sum_{i=0}^{n-1} (c_{i+1} * x^i)$ (p_n zal later gebruikt worden voor inductie).

$$C_{p_{n+1}} = \begin{bmatrix} 0 & 0 & \dots & 0 & -c_0 \\ 1 & 0 & \dots & 0 & -c_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & -c_{n-1} \\ 0 & 0 & \dots & 1 & -c_n \end{bmatrix}$$

$\Downarrow$ (karakteristieke veelterm)

$$\rho(C_{p_{n+1}}) = \begin{vmatrix} -\lambda & 0 & \dots & 0 & -c_0 \\ 1 & -\lambda & \dots & 0 & -c_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -\lambda & -c_{n-1} \\ 0 & 0 & \dots & 1 & -c_n - \lambda \end{vmatrix}$$

$\Downarrow$ (uitwerken naar eerste rij)

$$\begin{aligned} \rho(C_{p_{n+1}}) &= (-\lambda) * \begin{vmatrix} -\lambda & 0 & \dots & 0 & -c_1 \\ 1 & -\lambda & \dots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -\lambda & -c_{n-1} \\ 0 & 0 & \dots & 1 & -c_{n-1} - \lambda \end{vmatrix} + (-1)^n * (-c_0) * \begin{vmatrix} 1 & -\lambda & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -\lambda \\ 0 & 0 & \dots & 0 & 1 \end{vmatrix} \\ &= (-1)^n * \rho(C_{p_n}) + (-1)^{n+1} * c_0 \\ &= (-1)^n * (-\lambda) * \left(\lambda^n + \sum_{i=0}^{n-1} (c_{i+1} * \lambda^i) \right) + (-1)^{n+1} * c_0 \\ &= (-1)^{n+1} * \left(\lambda^{n+1} + \sum_{i=0}^{n-1} (c_{i+1} * \lambda^{i+1}) \right) + (-1)^{n+1} * c_0 \\ &= (-1)^{n+1} * \left(\lambda^{n+1} + \sum_{i=0}^{n} (c_i * \lambda^i) \right) \\ &= (-1)^{n+1} * p_{n+1}(\lambda) \end{aligned}$$

Aangezien de karakteristieke veelterm van de companionmatrix en de originele veelterm (desnoods op een teken na) aan elkaar gelijk zijn, zijn de wortels van beiden ook gelijk. Aangezien de wortels van de karakteristieke veelterm van de companionmatrix de eigenwaarden zijn, geldt het te bewijzen.

1.1.3 Sage-algoritme

De algoritmen voor deze opgave werden getest door middel van de testcode in appendix A.

Een eerste naïëve implementatie: op basis van het idee dat de nulwaarden van een veelterm niet veranderen wanneer we een veelterm vermenigvuldigen met een getal $c \in IR_0$. Hiermee kunnen wel elke veelterm p_n monisch maken door $p'_n(x) = \frac{p_n(x)}{c_n}$ te gebruiken (c_n is de hoogstegraadcoëfficiënt van p_n).

```
def companion_solve(poly):
    for i in poly:
        div = i
    mat = companion_matrix(poly / div)
    solutions = mat.eigenvalues()
    return solutions
```

Deze implementatie heeft nog een paar probleempjes: bij het berekenen van `poly / div` kunnen afrondingsfouten ontstaan (waardoor de companionmatrix niet meer berekend kan worden). Daardoor wordt een groot aantal van de testen overgeslagen. Daarnaast zitten er blijkbaar nog afrondingsfouten bij de berekening zelf, zoals te zien in de resultaten van de testen (voor elke graad, 10.000 testen gerund, met tolerantie `1e-10`):

Graad veelterm	Overgeslagen	Fouten	Slaagpercentage
2	1517	0	84.83% (100.0%)
3	1582	3	84.15% (99.96%)
4	1555	24	84.21% (99.72%)
5	1556	115	83.29% (98.64%)
6	1563	255	81.82% (96.98%)
7	1537	388	80.75% (95.42%)
8	1533	579	78.88% (93.16%)
9	1567	772	76.61% (90.85%)
10	1506	890	76.04% (89.52%)

Waarbij de slaagpercentages als volgt uitgerekend worden: $proc = \frac{aantal - overgeslagen - fout}{aantal}$ en tussen haakjes $proc' = \frac{aantal - overgeslagen - fout}{aantal - overgeslagen}$. Een manier om het foutpercentage te verminderen, is het gebruik van `RealField(100)` in de plaats van `RDF`. Dit vertraagt het volledige proces, natuurlijk. Het aantal overgeslagen testen verminderd helaas niet hiermee (tolerantie `1e-15`):

Graad veelterm	Overgeslagen	Fouten	Slaagpercentage
2	1601	0	83.99% (100.0%)
3	1534	0	84.66% (100.0%)
4	1510	0	84.90% (100.0%)
5	1486	0	85.14% (100.0%)
6	1502	2	85.15% (100.0%)
7	1462	1	84.96% (99.98%)
8	1511	7	84.82% (99.92%)
9	1562	6	84.32% (99.93%)
10	1535	23	84.42% (99.73%)

Het valt op dat het aantal veeltermen dat overgeslagen werd, ongeveer gelijk blijft.

Het overslaan vermijden: Nu worden er een zeer groot deel van de veeltermen overgeslagen door afrondingsfouten. Dit kunnen we vermijden door na de deling de coëfficënt bij de grootste macht van x standaard op 1 te zetten. Dit klopt omdat deze coëfficiënt altijd door zichzelf gedeeld wordt, en het resultaat van $\frac{n}{n}$ per definitie gelijk is aan 1. Deze aanpak resulteert in de volgende code:

```
def companion_solve(poly, ring=RDF):
```

```
tring.<x> = ring[]
c = (poly / poly.coefficients()[-1]).coefficients()
c[-1] = 1
poly2 = tring(c)
mat = companion_matrix(poly2)
solutions = mat.eigenvalues()
return solutions
```

Zoals verwacht worden er nu geen veeltermen meer overgeslagen:

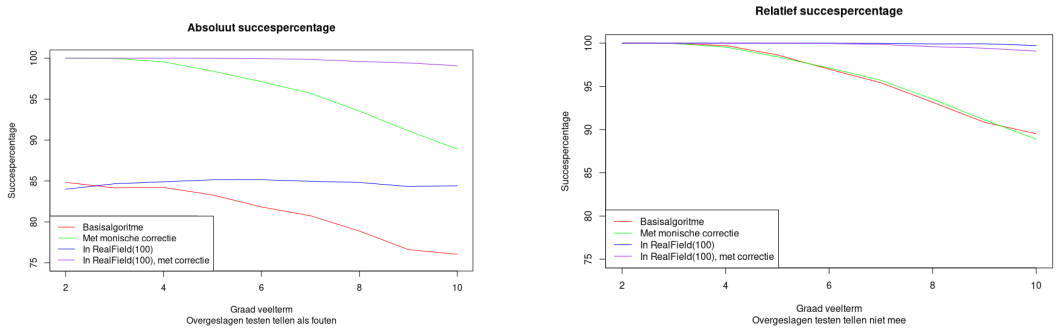
Graad veelterm	Overgeslagen	Fouten	Slaagpercentage
2	0	0	100.0% (100.0%)
3	0	3	99.97% (99.97%)
4	0	45	99.55% (99.55%)
5	0	158	98.42% (98.42%)
6	0	287	97.13% (97.13%)
7	0	429	95.71% (95.71%)
8	0	647	93.53% (93.53%)
9	0	887	91.13% (91.13%)
10	0	1111	88.89% (88.89%)

Als we bij het eerste algoritme de overgeslagen testen overlaten, krijgen we een gelijkaardige trend voor het percentage correcte testen. Door het wegwerken van de afrondingsfout bij de hoogstegraadseenterm, elimineren we een groot deel van de fouten (de overgeslagen testen, die anders mislukken). Natuurlijk zorgt ook hier voor een groter aantal bits (`RealField(100)`) voor een lager foutenpercentage (maar het opschalen van de precisie zorgt ook voor een veel tragere uitvoering):

Graad veelterm	Overgeslagen	Fouten	Slaagpercentage
2	0	0	100.0% (100.0%)
3	0	0	100.0% (100.0%)
4	0	0	100.0% (100.0%)
5	0	1	99.99% (99.99%)
6	0	4	99.96% (99.96%)
7	0	15	99.85% (99.85%)
8	0	40	99.60% (99.60%)
9	0	58	99.42% (99.42%)
10	0	92	99.08% (99.08%)

Opvallend is dat bij deze aanpak het aantal foute, niet-overgeslagen testen merkbaar groter is dan bij de versie zonder de coëfficiëntencorrectie. Hieronder een grafische vergelijking tussen de algoritmen:

Figure 1: Vergelijking algortimen



1.2 Methode van Durand-Kerner

Een eerste naïeve implementatie: door middel van de code voor de companionmatrix maken we eerst opnieuw de veelterm monisch, waarna we onderstaande berekening een aantal keer

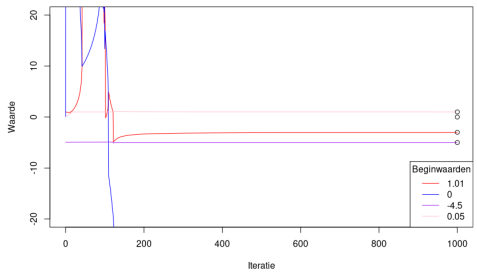
overlopen (bepaald door de parameter `iterations`).

$$x_i^{(n+1)} = x_i^{(n)} - \frac{p\left(x_i^{(n)}\right)}{\prod_{j\neq i}\left(x_i^{(n)} - x_j^{(n)}\right)}$$

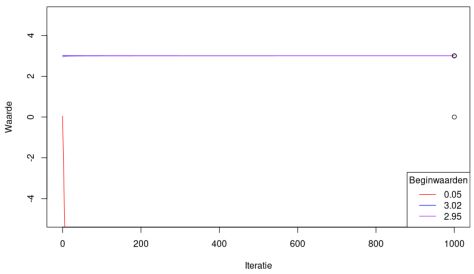
Figuur 2 bevat de resultaten voor twee (reëele) veeltermen. Een opvallend resultaat is dat het algoritme de nulwaarde 0 nooit vindt (ook niet bij complexe veeltermen). Voor testen met de (complexe) veeltermen $(x^3 - x^2 + (-5+2i)x + (-3-6i))$ en $(x^2 - (3+2i)x - (1-3i))$ zijn de resultaten erg gelijkaardig: snelle, lokale convergentie.

Figure 2: Resultaten

(a) $2x^4 + 14x^3 + 14x^2 - 30x$



(b) $2x^3 - 12x^2 + 18x$



Aanpassing noemer: tot nu toe werd de noemer $\left(\prod_{j\neq i}\left(x_i^{(n)} - x_j^{(n)}\right)\right)$ berekend in een for-loop. Dit geeft enkele problemen bij meervoudige nulwaarden: $x_i^{(n)}$ ligt dan erg dicht bij $x_j^{(n)}$, dus de noemer wordt erg klein. Indien we de list-comprehension (in combinatie met de `prod` functie)

```
prod([ 1 if elem == val else (val - elem) for elem in prev ])
```

gebruiken, worden deze gevallen al gedeeltelijk geëlimineerd.

Opvallend is wel dat deze methode minder dicht komt bij de effectieve nulwaarden; ze is meestal slechts tot op 2 of 3 beduidende cijfers correct. Opnieuw helpt de methode waarbij we van ring veranderen, maar dit zorgt ook niet voor de `1e-15` precisie die het andere algoritme haalde.

1.3 Sage default

Wanneer we naar de Sage source code kijken (op GitHub); meer specifiek naar het bestand src/sage/rings/polynomial/polynomial_element.pyx (lijn 7192) (de effectieve code start op lijn 7834), dan zien we dat het algoritme bepaald kan worden door de gebruiker (de parameter `algorithm`). Indien dit niet gebeurd, bepaalt Sage zelf het "beste" algoritme. De code is samen te vatten als volgt:

1. Indien de ring die de polynoom bevat, zelf een manier heeft om wortels te vinden, gebruik die manier (desnoods met gebruik van `algorithm`).

2. Indien `algorithm 'numpy'`, `'either'` of `None` is, gebruik de `numpy` algoritmen (deze geeft een warning wanneer we met getallen met meer dan 53 bits precisie werken). Volgens de numpy documentatie gebruikt dit de companionmatrix.

3. Indien het `numpy` algoritme faalt of `algorithm == 'pari'`, gebruik het `pari` algoritme. Volgens de pari documentatie gebruikt dit Schönhage's algoritme.

2 Littlewoodveeltermen

Om de C++ code te testen werd de JT++-suite gebruikt (zie de bestanden in de `/tests/`-directory).

2.1 Brute-force

2.1.1 Algoritme

CUDA/C++ implementatie: voor deze implementatie gebruikte ik Bairstow's Algoritme geïmplementeerd in C++ en geaccelereerd door middel van CUDA (op een GeForce GTX1050M). Bairstow's Algoritme berekent de nulwaarden door middel van factorisatie (elke veelterm met reële coëfficiënten kan ontbonden worden door factoren van de vorm $(x - a)$ en $(x^2 + bx + c)$ met negatieve discriminant). Hierdoor hoeven alleen berekeningen met reële getallen uitgevoerd te worden. De effectieve code is een variant op dit algoritme. Hieronder (algoritme 1, volgende bladzijde) staat de gebruikte pseudocode.

Voor de initiële schatting van de kwadratische coëfficiënten raadt Wikipedia aan om $r = \frac{a_{n-1}}{a_n}; s = \frac{a_{n-2}}{a_n}$ te gebruiken. Bij de Littlewoodveeltermen zullen deze waarden altijd -1 of 1 zijn. Als compromis gebruiken we $r = -1; s = 1$.

Optimalisaties tijdens het ontwerp: een aantal optimalisaties in de code konden al doorgevoerd worden tijdens het ontwerpen van de code zelf:

- De helft van de veeltermen hoeven niet opgelost te worden: $\forall p \in IR[x] : \forall x \in IR : p(x) = 0 \iff -p(x) = 0$. Dit heeft als voordeel dat een groot deel van de berekeningen niet opgelost hoeft te worden.
- Het genereren van de veelterm-arrays kan op de GPU gebeuren (parrallelisme). Hierdoor worden alle veeltermen gelijktijdig gegenereerd van hun index (op basis van de bits in hun index: een 0-bit stelt een coëfficiënt -1 voor, terwijl een 1-bit de coëfficiënt 1 voorstelt).

2.1.2 Praktische zaken

Moeilijkheden tijdens het schrijven: de originele berekening van r en s gebruikt een recursieve aanpak. Dit resulteerde echter in een `Segmentation Fault` (veroorzaakt door een stack-overflow). Ik kon dit oplossen door de recursieve call in het `if` blok op lijnen 26-29 te vervangen door een `while` loop. Een andere moeilijkheid werdt gevormd door de oneindige loop die ontstond; deze werd opgelost door na een bepaald aantal pogingen (in te stellen via de `MAX_TRIES` macro) de schatting voor de coëfficiënten willekeurig te stellen. Het implementeren van die willekeurige schattingen (op de GPU) bracht ook zijn eigen problemen mee, maar die werden relatief snel opgelost (dankzij Nicholas Shatokhin). Nadat alle individuele testen slaagden (1 veelterm), zat er nog een grove fout in de implementate: de schattingen van de coëfficiënten werden gedeeld tussen de threads. De oplossing hiervoor bestond ervan de schattingen de alloceren en initializeren in de initializatiefase van het algoritme. Dit zorgde voor de volgende initializatie:

Input: alle coëfficienten (a), graad v/d veelterm (g)

1 $modcoeff \leftarrow a[offset]$; `// offset op basis van GPU thread ID`
2 **if** $g \geq 1$ **then** $r \leftarrow \frac{modcoeff_{g-1}}{modcoeff_g}$;
3 **else** $r \leftarrow 1$;
4 **if** $g \geq 2$ **then** $s \leftarrow \frac{modcoeff_{g-2}}{modcoeff_g}$;
5 **else** $s \leftarrow 1$;

Algorithm 2: Initalizatie Bairstow's algoritme

Verder worden er nog een aantal parameters aangepast zodat alles GPU-gebaseerd kan gebeuren (en dus vele veeltermen tegelijk opgelost kunnen worden).

Andere startwaarden: wanneer het algoritme vastloopt (doordat de startwaarden niet convergeren), werden initieel gewoon nieuwe startwaarden gegenereerd. Dit gaf geen enkele garantie op convergentie. Een manier om dit probleem te proberen beperken was de willekeurige waarden te beperken binnen een bepaald interval. Dit werkte net iets beter, maar verhielp

Input: coëfficienten (a), graad v/d veelterm (g), beginwaarden (r, s)
Data: tolerantie (tol)
Output: array met nulwaarden

$result \leftarrow []$;
if $g < 1$ **then**
  return *result*;
end
if $g == 1$ **and** $a_1 \neq 0$ **then**
  $result.add(\frac{-a_0}{a_1})$;
  return *result*;
end
if $g == 2$ **then**
  $a' \leftarrow a_2, b \leftarrow a_1, c \leftarrow a_0$;
  $D \leftarrow b^2 - 4a'c$;
  $result.add(\frac{\sqrt{D}-b}{2a})$;
  $result.add(\frac{-\sqrt{D}-b}{2a})$;
  return *result*;
end
$n \leftarrow deg + 1$;
$b \leftarrow [0...0, a_{n-2} + r * a_{n-1}, a_{n-1}]$; // *deg* `elementen`
$c \leftarrow [0...0, b_{n-2} + r * b_{n-1}, b_{n-1}]$; // *deg* `elementen`
for $i \leftarrow n - 3$; $i >= 0$; $i - -$ **do**
  $b_i = a_i + r * b_{i+1} + s * b_{i+2}$;
  $c_i = b_i + r * c_{i+1} + s * c_{i+2}$;
end
$din \leftarrow (c_2^2 - c_3 * c_1)^{-1}$;
$r \leftarrow r + din * (-c_2 * b_1 + c_3 * b_0)$;
$s \leftarrow s + din * (c_1 * b_1 - c_2 * b_0)$
if $abs(b_0) > tol$ **or** $abs(b_1) > tol$ **then**
  $result.add(bairstow(a, g, r, s))$;
  return *result*;
end
if $g \geq 3$ **then**
  $dis \leftarrow r^2 + 4s$;
  $results.add(\frac{r-\sqrt{dis}}{2})$;
  $results.add(\frac{r+\sqrt{dis}}{2})$;
  $results.add(bairstow(b[2, ...], r, s, g - 2))$;
  return *result*;
end
return *result*;

Algorithm 1: Bairstow's algoritme

nog niet alles. Na het lezen van deze StackOverflow post, besliste ik om volgende waarden te gebruiken als nieuwe startwaarden:

$$\begin{aligned} rad &= \sum_{i=0}^{n} |a_i| \\ r &= 2 * \pi * rad^2 * cos(rng()) \\ s &= 2 * \pi * rad^2 * sin(rng()) \end{aligned}$$

Dit genereert de veelterm $x^2 + 2 * rad^2 * \pi * cos(\alpha)x + 2 * rad^2 * \pi * sin(\alpha)$. Het punt (r, s) ligt op de cirkel met straal $rad = \sum_{i=0}^{n} |a_i|$. Deze afstand, de "outer root radius", zorgt ervoor dat de waarden snel convergeren. Een vlugge test gaf een erg verrassend resultaat: voor een veelterm van graad n, werden exact n nieuwe paren startwaarden gegenereerd.

Testing: de bairstow code werd erg intensief getest. Een veelvoorkomende zwakte in algoritmen om veeltermen op te lossen zijn meervoudige nulwaarden. Deze toonden zich hier opnieuw een zwakte. De volgende testen zijn hardgecodeerd (in `tests/test_bairstow.cu`; met aangegeven veelterm):

- **Graad 0:** (het programma mag niet crashen)
- **Graad 1:**
  - 1 reële wortel: $3x - 7 = 0$ (wortel: $\frac{7}{3}$)
- **Graad 2:**
  - 2 reële wortels: $3x^2 - 10x - 8 = 0$ (wortels: 4 en $\frac{-2}{3}$)
  - 2 complexe wortels: $2x^2 + 14x + 25 = 0$ (wortels: $\frac{-7}{2} \pm \frac{1}{2}i$)
- **Graad 3:**
  - 2 complexe, 1 reële wortel: $x^3 - x^2 - 41x + 105 = 0$ (wortels: 7 en $\frac{-7}{2} \pm \frac{1}{2}i$)
  - 3 reële wortels: $x^3 - x^2 - 41x + 105$ (wortels: 3, 5 en -7)
- **Graad 4:**
  - 2 paar complexe wortels: $10x^4 - 12x^3 + 8x^2 - 2x + 1/2$ (wortels: $\frac{1}{10} \pm \frac{3}{10}i$ en $\frac{1}{2} \pm \frac{1}{2}i$)
  - 2 complexe, 2 reële wortels: $2x^4 - 27x^3 + 77x^2 + 263x - 795$ (wortels: $7 \pm 2i$, -3 en $\frac{2}{5}$)
  - 4 reële wortels: $84x^4 + 164x^3 - 608x^2 + 320x$ (wortels: $\frac{5}{7}$, -4, $\frac{4}{3}$ en 0)

Compilatie: om het programma te compileren worden volgende tools gebruikt (op Arch Linux):

- GNU make (om alles gemakkelijk en deel-per-deel te compileren)
- G++/GCC (voor de C++/niet-GPU code)
- NVCC (voor de GPU/CUDA code)

Running: na het volledige littlewood programa te compileren (door middel van `make`), kan het programma gestart worden met of zonder argumenten. De hoeveelheid argumenten is afhankelijk van de buildtime `PROFILING` flag:

- Zonder argumenten (`bin/bruteforce`) start het programma en vraagt de gebruiker om de graad van de littlewoodveeltermen. Enkel deze graad wordt gegenereerd en opgelost.
- 1 argument (`bin/bruteforce <graad>`; alleen wanneer `PROFILING` niet enabled is) start het programma en lost alle veeltermen van graad $[1, ..., graad[$ op. Indien *graad* 0 is, zal het programma zoveel mogelijk veeltermen oplossen.

- 2 argumenten (`bin/bruteforce <graad> <bestand>`) start het programma analoog aan de optie met 1 argument, maar zal, indien `PROFILING` enabled is, na elke iteratie de profilingdata naar het bestand schrijven.

Rendering: het littlewood-programma zelf (dmv de Makefile te compileren naar `bin/bruteforce`) schrijft de oplossingen naar `stdout`. Door deze stream om te leiden naar `res.txt`, verkrijgen we een dataset met platte tekst voor de resultaten (1 oplossing per lijn). Deze kunnen we door middel van een eenvoudig R script plotten:

```
data <- read.csv('/path/to/file', header=F, sep=' ')
plot(c(-2,2), c(-2,2), main="Littlewood Polynomial Roots",
     col='white', xlab='Re(x)', ylab='Im(x)')
points(data)
```

2.1.3 Toekomst

Adaptief renderen: eigenlijk zou een OpenGL programma in staat moeten zijn om deze resultaten dan (via stream of van een bestand) mooi zelf te renderen. Pogingen om dit te doen resulteerden helaas in vooral mislukkingen.

Verdere optimalisaties: de code op zich is nog niet optimaal. Een aantal dingen kunnen nog verbeterd worden:

- Een veelterm van graad n kan voorgesteld worden als een `int a` met $0 \leq a \leq 2^n$ ($n+1$ coëfficiënten, maar slechts n ervan hoeven gebruikt te worden, zie hierboven), waarbij de 0-bits gebruikt worden als coëfficiënt -1 en de 1-bits gebruikt worden als coëfficiënt 1.

2.1.4 Resultaten

Profiling: de littlewood programma bevat ingebouwde functionaliteit om de duur van de operaties te meten (indien de `PROFILING` preprocessor directive gedefinieerd is at buildtime).

2.2 Per-pixel

2.2.1 Bewijzen

In de bewijzen en eigenschappen wordt $L[x]$ gebruikt als de verzameling van de Littlewoodveeltermen, met andere woorden: $p(x) \in L[x] \iff (p(x) = \sum_{i=0}^{n} a_i x^i \wedge \forall i \in \{0, 1, ..., n\} : a_i \in \{-1, 1\})$. Verder wordt de notatie $z \in \mathbb{C}_{<1}$ gebruikt voor $z \in \mathbb{C} \wedge |z| < 1$.

1. $\underline{\forall z \in \mathbb{C}_{<1} : \forall l(x) \in L[x] : |l(z)| \leq \frac{1}{1-|z|}}$

   Bewijs:

$$
\begin{aligned}
|l(z)| &\leq l(|z|) && \text{(wegens l(x) is een som en } |a+b| \leq |a| + |b|) \\
&\leq l_{max}(|z|) && (|z| \text{ is een positief, reëel getal, dus } l_{max} = \sum_{i=0}^{\infty} x^i) \\
&= \lim_{t \to \infty} \sum_{i=0}^{t} |z|^i && \text{(met } 0 < |z| < 1) \\
&= \frac{1}{1-|z|} && \text{(reekssom van een meetkundige reeks)}
\end{aligned}
$$

2. $\underline{\forall z \in \mathbb{C} : (\exists l(x) \in L[x] : l(z) = 0) \implies (\exists l'(x) \in L[x] : l(z^{-1}) = 0)}$

Bewijs: (door constructie van l')

$$l(x) = \sum_{i=0}^{n} a_i x^i, \text{ met } l(z) = 0 \wedge a_0 \neq 0 \qquad \text{(def. littlewood)}$$

$$\text{neem } l'(x) = x^n * l(x^{-1}) = x^n * \sum_{i=0}^{n} a_i x^{-i} = \sum_{i=0}^{n} a_i x^{n-i}$$

$$\Downarrow$$

$$l'(z^{-1}) = (z^{-n}) * l((z^{-1})^{-1}) = (z^{-n}) * l(z) = (z^{-n}) * 0$$
$$= 0$$

3. $\forall(z, l(x)) \in \mathbb{C}_{<1} \times L[x], d = gr(l(x)) : (|l(z)| > \frac{|z|^{d+1}}{1-|z|})$
$\overline{\implies \neg(\exists l'(x) \in L[x] : l' \text{ is een uitbreiding van } l \wedge l'(z) = 0)}$

Bewijs: (uit het ongerijmde; stel $\exists l' \in L[x] : l'(z) = 0$)

$$0 = l'(z) = l(z) + \sum_{i=d+1}^{n} c'_i z^i = \sum_{i=0}^{d} c_i z^i + \sum_{i=d+1}^{n} c'_i z^i$$

$$\Updownarrow$$

$$\sum_{i=0}^{d} c_i z^i = -\sum_{i=d+1}^{n} c'_i z^i$$

$$\Downarrow$$

$$\frac{|z|^{d+1}}{1-|z|} \overset{(1)}{<} \left|\sum_{i=0}^{d} c_i z^i\right| = \left|\sum_{i=d+1}^{n} c'_i z^i\right|$$

$$\Updownarrow$$

$$\frac{|z|^{d+1}}{1-|z|} < \left|\sum_{i=d+1}^{n} c'_i z^i\right| \overset{(2)}{\leq} \sum_{i=d+1}^{n} \left|c'_i z^i\right| \overset{(3)}{=} \sum_{i=d+1}^{n} \left|z^i\right| \overset{(4)}{=} \sum_{i=d+1}^{n} |z|^i$$

$$\frac{|z|^{d+1}}{1-|z|} < \sum_{i=d+1}^{n} |z|^i$$

$$\Updownarrow$$

$$|z|^{d+1} \leq (1-|z|) * \sum_{i=d+1}^{n} |z|^i = \sum_{i=d+1}^{n} |z|^i - \sum_{i=d+1}^{n} |z|^{i+1}$$
$$= |z|^{d+1} + (|z|^{d+2} - |z|^{d+2}) + \ldots + (|z|^n - |z|^n) - |z|^{n+1}$$
$$= |z|^{d+1} - |z|^{n+1}$$

$$\Updownarrow (5)$$

$z = 0 \ \sim$ Tegenstrijdigheid: $z = 0$ is nooit een oplossing voor een $l_n(x) \in L[x]$

Gebruikte hulpstellingen:

1) $l(x) > \dfrac{|z|^{d+1}}{1-|z|}$ (gegeven)

2) $\forall c \in \mathbb{C} : |a+b| \leq |a| + |b|$

3) $c_i \in \{-1, 1\} \implies (c_i z^i) \in \{-z^i, z^i\} \implies \left|c_i z^i\right| = \left|z^i\right|$

4) $\left|z^i\right| = \left|\prod_{j=1}^{n} z\right| = \prod_{j=1}^{n} |z| = |z|^i$

5) $|z|^a \geq 0 \implies (|z|^{d+1} = |z|^{d+1} - |z|^{n+1} \iff z = 0)$

2.2.2 Algoritme

Behalve de gegeven stellingen, werd ook gebruik gemaakt van stellingen uit Majken Roelfszema's thesis, meer bepaald: (haar thesis maakt gebruik van volgende notatie: $D = \{z | z \in \mathbb{C} \wedge \exists l(x) \in L[x] : l(z) = 0\}$)

1. **Stelling 1:** $\forall z \in D : \frac{1}{2} < |z| < 2$ (met D de verzameling van alle nulwaarden van alle littlewoodveeltermen.
2. **Stelling 2:** D is symmetrisch in de reële as, de imaginaire as en de eenheidscirkel.
3. **Stelling 20:** $\{z \in \mathbb{C}, |z| \in [\frac{1}{\sqrt[4]{2}}, \sqrt[4]{2}]\} \subset \overline{D}$, waarbij $\overline{D}$ ook de oplossingen van $\lim_{deg(l)\to\infty} l(x)$ ($l(x)$ is een littlewoodveelterm) bevat.

Met behulp van deze stellingen kan een relatief eenvoudig algoritme uitgewerkt worden waarbij eerst een paar "basisgevallen" overlopen worden, waarna een eenvoudige `while`-loop zal proberen om een veelterm op te stellen die zichzelf corrigeert richting 0: voor elke volgende graad kunnen slechts 2 coëfficiënten toegevoegd worden aan de veelterm: 1 of -1. De coëfficiënt die de ervoor zorgt dat de modulus van het resultaat het kleinst wordt, zal opnieuw gebruikt worden in de volgende iteratie. De lus stopt op het moment dat een oplossing is gevonden, de voorwaarde uit het laatste bewijs waar wordt voor beide coëfficiënten, of de volgende term kleiner wordt in modulus dan de tolerantie. Met andere woorden, het algoritme probeert (beginnend van de veelterm $l(x) = 1 \in L[x]$) telkens zoveel mogelijk te corrigeren richting 0. Dit is mogelijk als en slechts als geldt dat $0 \leq |z| < 1$. Om dit te doen kloppen, controleren we of in $\frac{1}{z}$ een veelterm gegenereerd kan worden. Door middel van de stellingen 1 en 20 uit de gelinkte thesis, kunnen we relatief eenvoudig voor een groot aantal punten al bepalen of ze al dan niet kunnen bestaan.

Een initiële optimalizatie: de waarde $|a+bi|$ wordt in het algoritme meermaals berekend. Aangezien $|a+bi| = \sqrt{a^2+b^2}$ een vierkantswortel bevat (en dit een dure operatie is), zullen we in de praktijk de wortel zoveel mogelijk proberen wegwerken. Bij de meeste vergelijkingen kunnen we gewoon beide leden kwadrateren, omdat met constanten gewerkt wordt. Voor de vergelijkingen in lijn 23 ($|low| > norm \wedge |high| > norm$; met $low = c+di, high = e+fi, norm = \frac{|a+bi|^{d+1}}{1-|a+bi|}$) is dit net iets minder eenvoudig (een analoge afleiding kan gebruikt worden voor $high > norm$):

$$
\begin{aligned}
low > norm &\iff |c+di| > norm \\
&\iff \sqrt{c^2+d^2} > \frac{\sqrt{a^2+b^2}^{d+1}}{1-\sqrt{a^2+b^2}} \\
&\iff c^2+d^2 > \frac{\sqrt{a^2+b^2}^{2(d+1)}}{\left(1-\sqrt{a^2+b^2}\right)^2} = \frac{\left(a^2+b^2\right)^{d+1}}{1-2\sqrt{a^2+b^2}+a^2+b^2}
\end{aligned}
$$

Een laatste plaats waar de wortel niet weggewerkt kan worden is in de uitdrukking $\frac{1}{z} = \frac{1}{a+bi} = \frac{a}{|a+bi|} + \frac{-b}{|a+bi|}i = \frac{a}{\sqrt{a^2+b^2}} + \frac{-b}{\sqrt{a^2+b^2}}i$.

Uitzondering: indien $|z| = 1$ zal dit algoritme over het algemeen niet convergeren. Door middel van stelling 7 uit M. Roelfszema's thesis ($\forall r \in \mathbb{Q} : e^{i\pi r} \in D$) kunnen we stellen dat elk punt met $1 - tol \leq |z| \leq 1 + tol$ op de eenheidscirkel ligt en er een $r \in \mathbb{Q}$ bestaat waarvoor $z = e^{i\pi r}$; wat dus betekent dat elk punt op de eenheidscirkel kan meetellen als nulwaarde.

Input: Coordinate of the OpenGL fragment $((x, y))$
Data: tolerance for the algorithm (tol)
Output: True if $\exists l(x) \in L[x] : l(x + yi) = 0$, otherwise False

$a \leftarrow x$, $b \leftarrow y$;
if $|a + bi| \leq \frac{1}{2} \vee |a + bi| \geq 2$ **then**
  return *False*; `// Theorem 1 from the linked thesis`
end
if $a < 0$ **then**
  return *isSolution(-a, b)*; `// Theorem 2 from the linked thesis`
end
if $b < 0$ **then**
  return *isSolution(a, -b)*; `// Theorem 2 from the linked thesis`
end
if $|a + bi| > 1$ **then**
  return $isSolution(\frac{a}{|a+bi|}, \frac{-b}{|a+bi|})$; `// Theorem 2.2.1.2`
end
$curr \leftarrow a + bi, px \leftarrow 1, norm \leftarrow \frac{|a+bi|}{1-|a+bi|}$;
while $|curr| > tol$ **do**
  $low \leftarrow px - curr, high \leftarrow px + curr$;
  if $|low| \leq tol \vee |high| \leq tol$ **then**
    return *True*; `// We found a polynomial`
  end
  if $|low| > norm \wedge |high| > norm$ **then**
    return *False*; `// Neither way will get a solution`
  end
  $curr \leftarrow curr * (a + bi), norm \leftarrow norm * |a + bi|$;
  if $|low| < |high|$ **then**
    $px \leftarrow low$
  end
  else
    $px \leftarrow high$
  end
end
return *False*; `// The result doesn't change anymore`

Algorithm 3: $isSolution(x, y)$

2.2.3 Hardware limitatie

Huidige (NVIDIA) GPUs ondersteunen geen `branch prediction`, met andere woorden, elke conditionele jump is een enorm dure operatie op een GPU. Dit zorgt ervoor dat de `while`-loop uit de pseudocode enorm inefficiënt en traag is. Indien het aantal iteraties tijdens het compileren al gekend is, kan de compiler een techniek toepassen die `loop unrolling` heet. Deze techniek bestaat eruit elke jump te elimineren door elke iteratie lineair na elkaar in de bytecode te zetten. Het aantal iteraties dat een $z \in \mathbb{C}, |z| < 1$ nodig heeft om gelijk aan tol te worden, kan eenvoudig berekend worden:

$$|z^n| = tol \iff |z|^n = tol \iff n = \log_{|z|} tol$$

Met andere woorden, het aantal iteraties is afhankelijk van de tolerantie (het is ook diezelfde tolerantie die bepaald of z veel verandert of niet). Om een gangbare waarde te vinden werd het algoritme ook eens op de CPU uitgevoerd, waarbij het aantal iteraties gelogd werd (elke keer werden 1024x1024 punten berekend). Uit deze plots blijkt dat de complexe getallen die het meeste

Figure 3: Aantal iteraties (CPU) $z \in \{a + bi | a, b \in [-2, 2]\}$

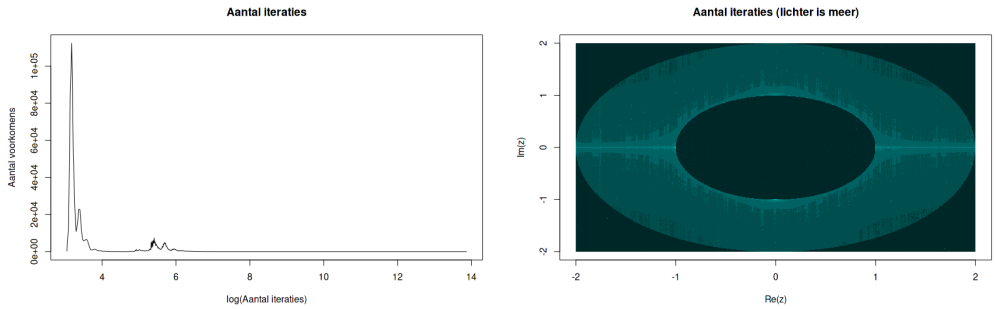
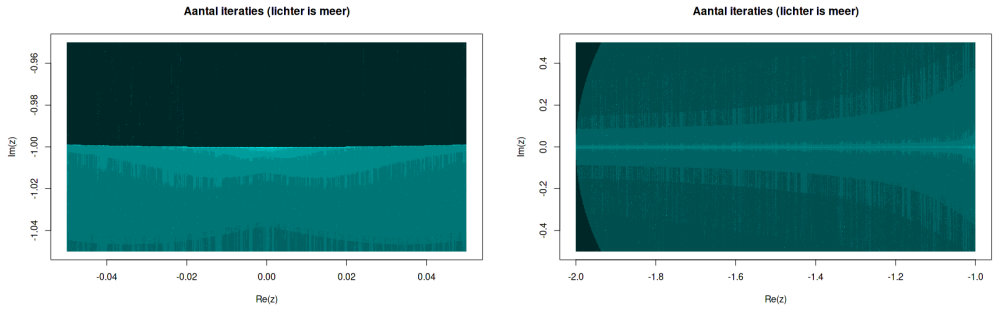


Figure 4: Aantal iteraties (CPU, ingezoomd)



iteraties nodig hebben, zich bevinden op de assen, dicht bij $|z| \sim 1$. Om tot een goede schatting te komen, werd eens ingezoomd op de gebieden $z \in \{a + bi | a \in [-0.05, 0.05], b \in [-1.05, -0.95]\}$ en $z \in \{a + bi | a \in [-2, -1], b \in [-0.5, 0.5]\}$.

Aangezien dit aantal erg groot kan worden, zal deze shader heel snel heel inefficiënt en traag worden. We kunnen hierrond door CUDA en OpenGL samen te laten werken, waarbij de CUDA kernel zijn resultaten zal "renderen" naar een texture, en OpenGL deze texture zal gebruiken bij het renderen van een fragment. Deze texture zal 4096 horizontale en 4096 verticale samples bevatten zodat de OpenGL context nooit zal moeten extrapoleren tijdens het renderen. Hierdoor ziet de uiteindelijke render loop er ruwweg als volgt uit:

```
Set up OpenGL;
Link OpenGL texture to CUDA;
while Not yet exiting do
    Handle input;
    if Input changed position/scale then
        Use CUDA kernel to generate texture;
        Synchronize GPU;
    end
    Set up frame (OpenGL calls);
    Swap OpenGL buffers;
end
```

Algorithm 4: Render loop/driver code

Om deze complexe samenwerking tot een goed einde te brengen, werd gebruik gemaakt van Forceflow/cuda2GLcore. De uiteindelijke implementatie kan arbitrair snel (of arbitrair exact) werken, enkel gelimiteerd door hardware en de IEEE `float` specificatie.

2.2.4 Uiteindelijke applicatie

Compilatie: net zoals bij vorig deel is ook hier een Makefile meegeleverd zodat de code snel en eenvoudig gebouwd kan worden. De code heeft dependencies op:

1. GNU Make (om deel per deel te compileren)
2. NVCC (de NVIDIA compiler), en G++ (om niet-CUDA code te compileren)
3. CUDA en de CUDA SDK (`cuComplex`) voor de eigenlijke berekeningen
4. OpenGL ($\geq$ 4.6 Core), GLEW, GLFW (voor het renderen). Hierbij horen ook de X11 libraries (X11, Xrandr, Xi).

Controls: wanneer de applicatie niet aan het rekenen is, kunnen de volgende toetsen gebruikt worden om de camera te besturen. Indien de camera aangepast is, zal het frame erna de resulterende afbeelding opnieuw berekend worden.

1. *W, S:* beweeg de camera naar boven, resp. onder
2. *A, D:* beweeg de camera naar links, resp. rechts
3. *Q, E:* zoom uit, resp. in
4. *Spatie:* reset de camera
5. *Escape:* sluit de applicatie

Het beeld toont elke frame op de linkerzijde (in het rood) de verzameling D (de oplossingen) en op de rechterzijde (in het blauw) het aantal iteraties dat nodig was om tot die oplossing te komen. Om de complexe getallen te berekenen die overeenkomen met de pixels wordt volgende formule gebruikt (*zoom* en *center* zijn eigenschappen van de camera, waarbij *zoom* groter wordt naarmate verder ingezoomd wordt en waar *center* het complex getal op de middelste pixel voorstelt).

$$\mathcal{T} : [0, w] \times [0, h] \to CamSpace :$$
$$(x, y) \to \left(\frac{4 * x}{w * zoom} - \frac{2}{zoom} + Re(center), \frac{4 * y}{h * zoom} - \frac{2}{zoom} + Im(center)\right)$$

Snelheid: zoals eerder vermeld is de snelheid van de implementatie afhankelijk van twee factoren: het maximaal aantal iteraties en de precisie. Onder "basisomstandigheden" (2^{20} iteraties, precisie $1 * 10^{-20}$, standaardcamera), haalt de applicatie ongeveer een 0.2 fps (dus ongeveer 5s om de volledige afbeelding te renderen), indien de kernel berekend moet worden. Onder idle (geen aanpassingen in de camera), haalt de applicatie een 60-tal FPS (dit is het harde limiet dat door GLFW wordt opgelegd). De implementatie werd ook geprofileerd bij andere instellingen. Het

profileren kan opnieuw aangepast worden door middel van macros (`PROFILING` om de profilering aan of uit te zetten. Indien `#if PROFILING` worden de precisies in [`MIN_PRC`, `MAX_PRC`] getest door telkens de precisie met `DELTA_PRC` te vermenigvuldigen; het aantal iteraties kan analoog aangepast worden door middel van `MIN_IT`, `MAX_IT` en `DELTA_IT`; elke combinatie van precisie en iteraties word `REPEATS` keer getest).

In figuren 5 en 6 wordt deze data geplot, waarbij elke kleur zijn eigen betekenis heeft; groen stellen de minima voor, grijs de gemiddelden, blauw de medianen en rood de maxima. Alle grafieken gebruiken een logaritmische schaal (met basis 10 voor de precisie en basis 2 voor het aantal iteraties) om de data mooi over de x-as te verdelen. Deze basissen zijn zo gekozen omdat dit ook gereflecteerd wordt in de waarden (het aantal iteraties evolueerde tijdens het genereren van de data door middel van bitshifts (`1 << x`), terwijl voor de precisie telkens de exponent in de wetenschappelijke notatie aangepast werd (`1e-x`)).

Telkens wordt één van beide parameters genomen om te vergelijken met een paar waarden van de andere, om zo te kunnen uitzetten tegenover de fps.

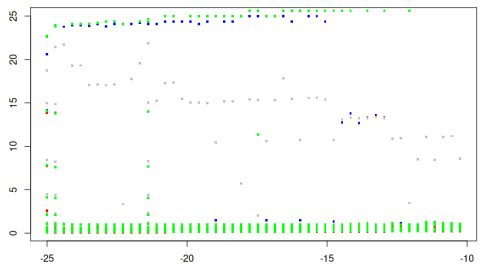
Wanneer de precisie uitgezet wordt, blijkt dat deze ofwel erg grillig is, ofwel relatief constant. Enkel wanneer het aantal iteraties erg groot wordt, krijgen we een meer "verwachtte" grafiek: de fps groeit naarmate het de precisie daalt (naarmate de threshold groter wordt).

Het aantal iteraties toont een heel ander beeld: de grafiek is telkens erg gelijkaardig, met waarden die (zo goed als) altijd erg dicht bij elkaar liggen (de meeste lijnen overlappen).

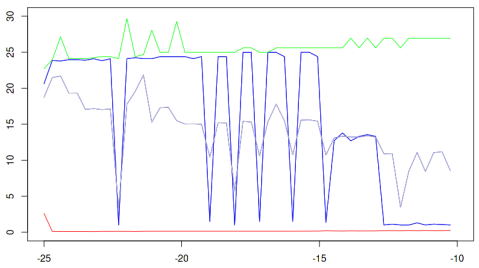
Figuur 7 zet alle gegevens tegenover elkaar uit: op de x-as vinden we telkens de precisie terug (in $\log_{10}$ om een gelijkmatige spreiding te bekomen), op de y-as het maximaal aantal iteraties (opnieuw in $\log_2$) en de behaalde fps wordt (als $\log_2$) gekleurd waarbij rood laag is, blauw gemiddeld en groen hoog (relatief gezien). Deze figuren tonen aan dat de bepalende factor niet zozeer de precisie als wel het maximaal aantal iteraties is.

Figure 5: FPS/Precisie-plots ($\log_{10}(prec)$)

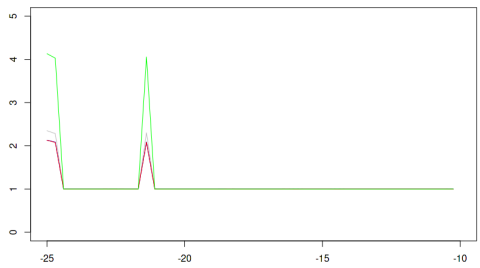
(a) Overzicht van alle statistieken



(b) Bij 1024 (1 << 10) iteraties



(c) Bij 16384 (1 << 14) iteraties



(d) Bij 33554432 (1 << 25) iteraties

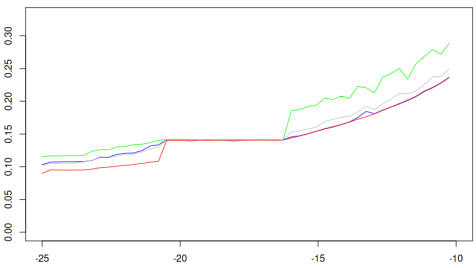
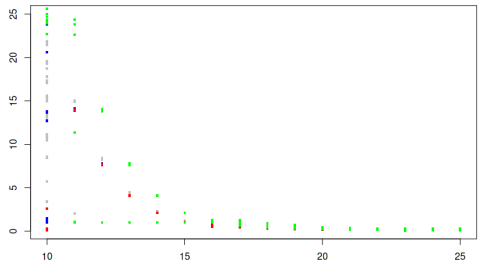
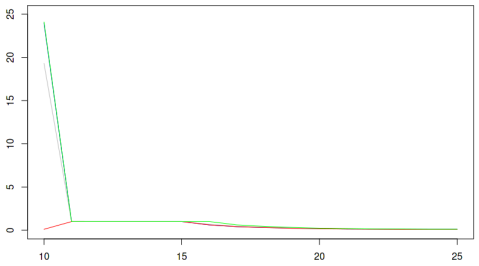


Figure 6: FPS/maximaal aantal iteraties ($\log_2(it)$)

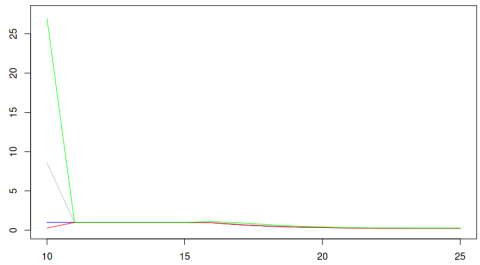
(a) Overzicht van alle statistieken



(b) Bij precisie $8 * 10^{-25}$



(c) Bij precisie $5.6295 * 10^{-11}$



(d) Bij precisie $2.14748 * 10^{-16}$

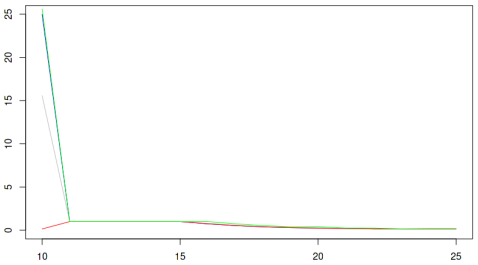
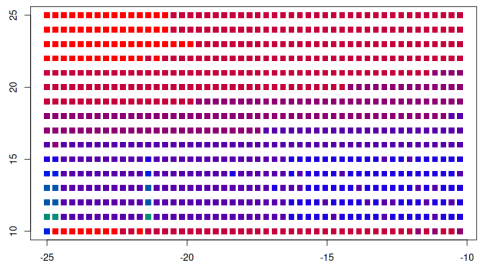
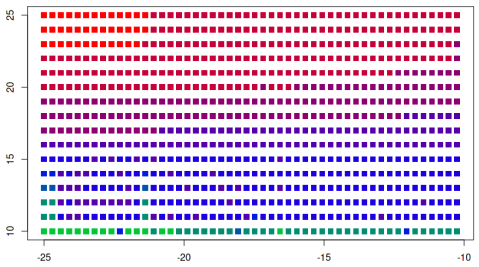


Figure 7: FPS t.o.v. beide parameters; $\log_2(fps)$, $x = \log_{10}(prec)$, $y = \log_2(it)$

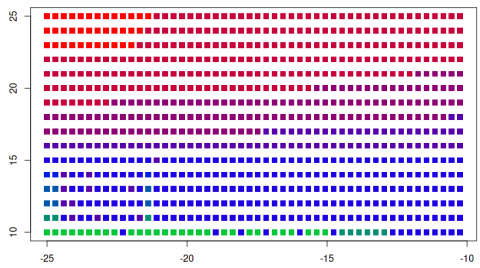
(a) Minimum



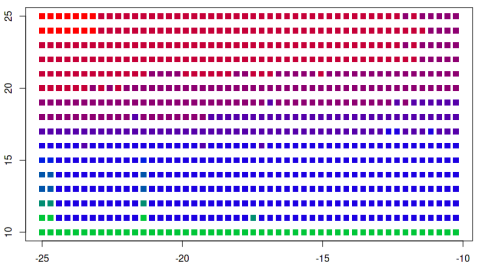
(b) Gemiddelde



(c) Mediaan



(d) Maximum



Vergelijkende screenshots: hieronder (figuren 8, 9 en 10) staan een aantal paren screenshots van bepaalde (deel)gebieden van het complexe vlak. Elk paar bevat een screenshot van de wortels (rood, op links) en een screenshot van het aantal iteraties per punt in het complexe vlak (blauw, op rechts).

Figure 8: Zoom 1.0, gecentreerd rond $z = 0$

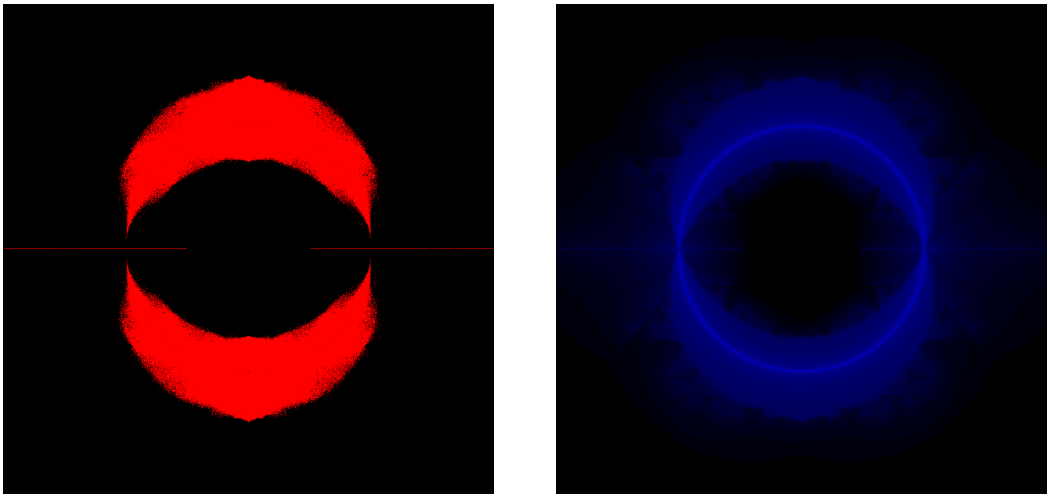


Figure 9: Zoom 3.13843, gecentreerd rond $z = \frac{1}{2} + \frac{1}{2}i$

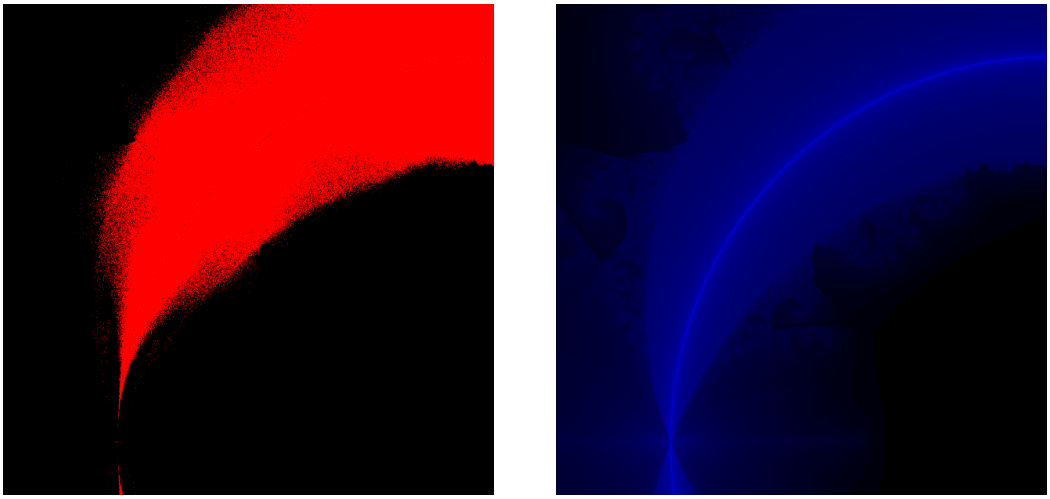
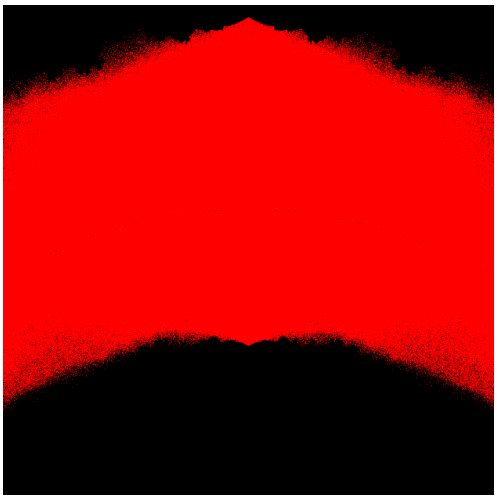
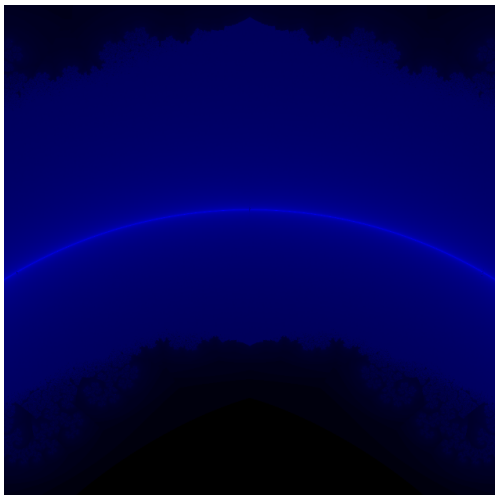


Figure 10: Zoom 3.7975, gecentreerd rond $z = -0.0658328 + 0.914028i$





A Testcode Sage

Deze code werd gebruikt om de Sage-code voor opgave 1.1 (nulpunten van (monische) veeltermen door middel van de companionmatrix) te testen. Voor elke graad van 2 tot en met 10 worden er 10000 (willekeurige) veeltermen van die graad berekend. Het algoritme wordt gebruikt om de nulwaarden te zoeken, waarna de functiewaarden van deze nulwaarden vergeleken wordt met een vaste drempelwaarde ($1 * 10^{-15}$)

```
def test_solve(func, poly, margin=1e-15, ring=RDF):
    try:
        sol = func(poly, ring)
    except(ValueError):
        return (True, 0)
    for ans in sol:
        a = abs(poly(ans))
        if a > margin:
            return (False, ans)
    return (True, -1)

def multitest_solve(func, degree=5, samples=10000, margin=1e-15):
    #P.<x> = RDF[]
    P.<x> = RealField(100)[]
    errors = 0
    skipped = 0
    for i in range(samples):
        p = P.random_element(degree)
        (res, err) = test_solve(func, p, margin, RealField(100))
        if res == False:
            errors += 1
        if res == True and err == 0:
            skipped += 1
    print("Ran ", samples - skipped, " tests for polynomials of degree ",
        degree, " (", skipped, " skipped); ", errors,
        " solutions outside of margin ", margin)
```

B Directory-structuur

Hierbij een kort overzicht van wat waar te vinden is.

- `./bruteforce-cuda/`: bevat de code voor de brute-krachtsimplementatie (C++/CUDA implementatie).
  - `./bruteforce-cuda/bin/`: wordt gebruikt als output directory voor binaries.
  - `./bruteforce-cuda/obj/`: wordt gebruikt als objects directory.
  - `./bruteforce-cuda/src/`: bevat de C++/CUDA source code.
  - `./bruteforce-cuda/tests/`: bevat de C++/CUDA code voor de tests.
  - `./bruteforce-cuda/Makefile`: de Makefile om het bruteforce gedeelte eenvoudig te compileren.
- `./elegant-gl/`: bevat de code voor de elegantere implementatie (C++/CUDA/OpenGL implementatie).
  - `./elegant-gl/bin/`: wordt gebruikt als output directory voor binaries.
  - `./elegant-gl/datasets/`: bevat een aantal figuren die gebruikt worden in dit document.
  - `./elegant-gl/glad/`: bevat de GLAD source code (gebruikt om OpenGL te initializeren; gedownload van https://glad.dav1d.de/.
  - `./elegant-gl/obj/`: wordt gebruikt als objects directory.
  - `./elegant-gl/scrsh/`: bevat de screenshots voor 2.2.4.
  - `./elegant-gl/shaders/`: bevat de OpenGL vertex shader en de twee OpenGL fragment shaders.
  - `./elegant-gl/src/`: bevat de C++/OpenGL source code.
    - `./elegant-gl/src/cuda/`: bevat de C++/CUDA source code.
  - `./elegant-gl/Makefile`: de Makefile om het elegantere gedeelte eenvoudig te compileren
- `./figures/`: bevat een deel van de figuren in dit bestand.
- `./notebook.ipynb`: het Sage/Jupyter Notebook voor opgave 1.
- `./verslag.pdf`: (dit bestand) het verslag.

C Makefile targets

Hieronder volgt per Makefile in het project een kort overzicht van elke recept (target) in die Makefile.

C.1 Bruteforce (./bruteforce-cuda/Makefile)

- `all` (default): compileert en linkt alle code voor `./bruteforce-cuda/bin/bruteforce`; de brute-krachtsimplementatie die alle oplossingen poogt te berekenen.
- `cuda`: synoniem voor `all`
- `test`: compileert en linkt alle testen, en voert die daarna ook uit.
- `clear`: verwijdert alle binaries en/of objecten, en zet daarna de directory structuur klaar voor een volgende compilatie.

C.2 Elegant (./elegant-gl/Makefile)

- `all` (default): compileert en linkt alle code voor `./elegant-gl/bin/gl`; de elegantere implementatie die per pixel probeert een veelterm op te stellen, en daarbij ook het aantal benodigde iteraties weergeeft.
- `gl`: synoniem voor `all`.
- `cpu`: bouwt een eenvoudige (CPU) implementatie in . Deze implementatie werd gebruikt voor (de figuren in) 2.2.3.
- `clear`: verwijdert alle binaries en/of objecten, en zet daarna de directory structuur klaar voor een volgende compilatie.