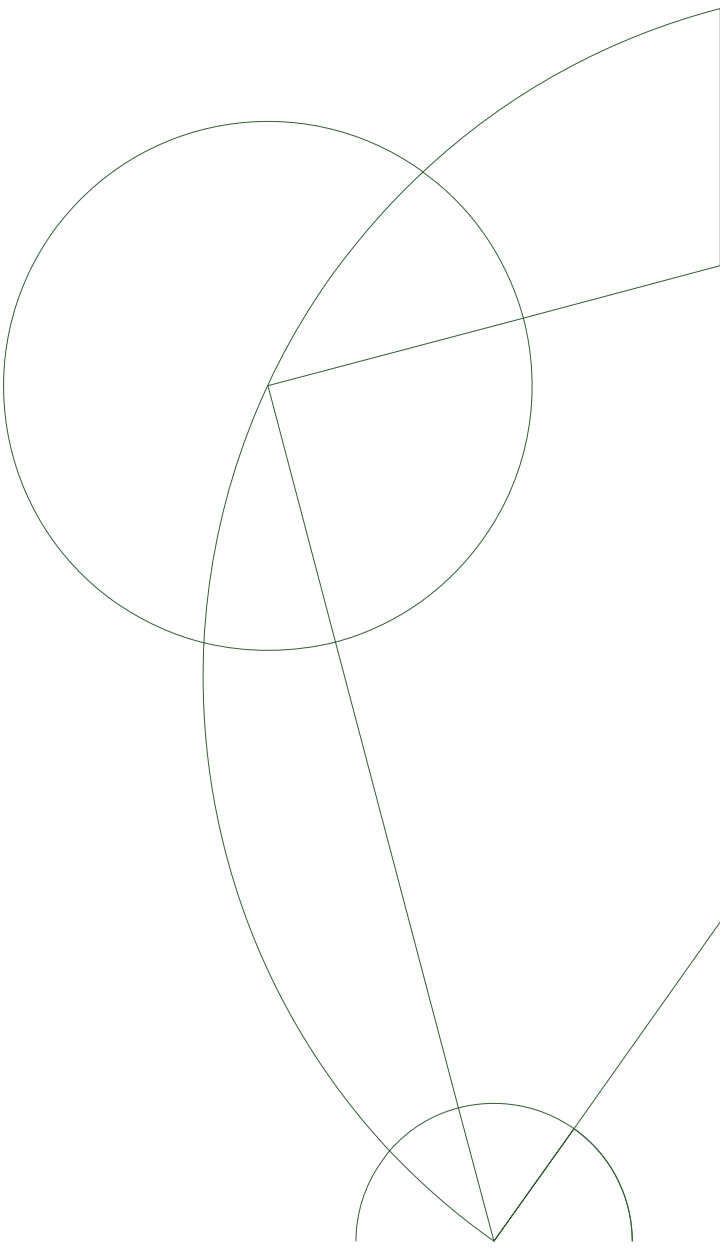
Kandidatspeciale, Datalogisk Institut
Københavns Universitet, december 2008
Vejleder Jyrki Katajainen

Robusthed i geometriske algoritmer

Michael Neidhardt



Abstract

The description of many geometric algorithms assumes that all arithmetic computations produce correct results. Naively implementing an algorithm with floating point-types exposes the program to rounding errors, which can lead to qualitative errors and program crashes. These are called robustness problems and several solutions exist. In this report I analyse a number of algorithms that solve the all-nearest-neighbors-problem and analyse a number of solutions to robustness problems, notably semi-static floating point filters and multiprecision types. The algorithms have been implemented with and without robust solutions. Benchmarks confirm the overall picture that robustness comes at a price, that filters are superior to multiprecision types, and that it is relatively hard to provoke this kind of error, meaning that it is often sufficient to use floating point calculations. The last fact makes filters attractive, in that they often use floating point-calculations as the first step.

Resumé

I beskrivelsen af mange geometriske algoritmer antages det at alle beregninger giver et korrekt resultat. Ved en naiv implementering med flydende tal risikerer man dog afrundingsfejl. Disse fejl kan medføre andre fejl, også kvalitative, som kan bringe et program i en udefineret tilstand og få det til at gå ned. Sådanne problemer kaldes robusthedsproblemer. Der er forskellige måder at afhjælpe dem på. Jeg analyserer i denne rapport en række algoritmer der løser det geometriske problem alle-nærmeste-naboer. Jeg analyserer et antal løsninger på robusthedsproblemer, primært semistatiske filtre og multipræcisionstyper og har implementeret algoritmerne både med og uden disse løsninger. En række testkørsler bekræfter alle det overordnede billede, nemlig at robusthed koster i form af ekstra køretid, at filtre er multipræcisionstyper overlegne, og at det er relativt svært at fremprovokere fejl, hvilket vil sige at man i mange tilfælde kan nøjes med beregninger med flydende tal. Det sidste gør filtre attraktive, idet de ofte har beregning med flydende tal som første trin.

Indhold

1

Introduktion

Denne rapport dokumenterer mit forsøg på at undersøge og beskrive robusthed i geometriske algoritmer. At en algoritme er robust vil her sige at den regner korrekt, at den derfor leverer et korrekt svar, og at den stopper. At dette overhovedet er et problem skyldes primært afrundingsfejl. Geometriske algoritmer er oftest designet til beregningsmodellen kaldet *reel RAM* (RRAM) og med inddata i *generel position* [17, 40, 43]. RRAM er en model hvori de primitive operationer (addition, subtraktion, multiplikation, division, sammenligninger) antages anvendt på reelle tal med uendelig præcision - se [38]. At inddata er i generel position betyder fx i planen at der hverken findes tre punkter på samme linie eller fire punkter på samme cirkel. Når en sådan algoritme implementeres benyttes ofte flydende tal, som kun er en tilnærmelse til de reelle, med deraf følgende risiko for afrundingsfejl. Det er herudover en udbredt opfattelse at geometriske algoritmer er ikke-trivielle at implementere — se [17, 30, 31].

1.1 Casestudie

Jeg undersøger emnet ved at analysere, implementere og afprøve en række algoritmer. De løser alle det geometriske problem der hedder *alle nærmeste naboer, ANN*. Givet en punktmængde S indeholdende n punkter, og en metrik, der angiver afstand mellem punkterne i S, kan ANN-problemet defineres som: For ethvert punkt $p \in S$ find et punkt i $S \backslash \{p\}$ der er tættest på p. Som antydet findes en række forskellige algoritmer der løser det. I dette speciale begrænser jeg mig til løsninger i to dimensioner. Se [15] for en mere komplet gennemgang af algoritmer til løsning af ANN. De algoritmer jeg ser på er inkrementel Delaunay-triangulering (to udgaver), *plane sweep* og Kd-træer. De valgte algoritmer bliver beskrevet og jeg gør detaljeret rede for de dele der er direkte relateret til robusthed og eksakthed.

1.2 Geometriske primitiver

Geometriske primitiver som at bestemme hvor et punkt ligger i forhold til en linie gennem to andre punkter, og at bestemme hvorvidt et punkt er på, uden for eller inde i en cirkel gennem tre andre punkter er meget udbredte i geometriske algoritmer. En funktion som her kaldes *orient2d* afgør det første spørgsmål, og

en anden funktion kaldet *incircle* afgør det andet. I begge indgår beregning af fortegnet for en determinant. På engelsk kaldes disse funktioner for *predicates* hvilket jeg oversætter til *prædikater* på dansk. Andre funktioner danner nye objekter, fx skæringspunktet mellem to ikke-parallelle linier i planen, og disse kaldes konstruktorer. Svarene fra disse prædikater, og objekterne der dannes i konstruktorerne bestemmer i nogle tilfælde strukturen (topologien) for ens løsning, og altså kan en kvantitativ fejl medføre kvalitative fejl. Eksemplerne er mange, fx trianguleringer i planen der er ikke-planare, konvekse hylstre der ikke er konvekse, programmer der går ned o.l. (se fx [32, 34, 35, 40, 43]).

1.3 Heltal eller flydende tal

Det er oplagt at unøjagtigheder forårsaget af afrundingsfejl er forbundet med flydende tal, og ikke heltal, men løsningen er desværre ikke bare at benytte heltal. Det var ikke fra starten af mit arbejde med denne opgave fastlagt om inddata skulle begrænses til den ene eller den anden type, men i og med at beskrivelsen af en af algoritmerne forudsætter heltallige koordinater blev det fastslået at heltal også ville være relevante.

Metoder som beskrevet i [41] arbejder med flydende tal, og forfatteren gør sig også overvejelser om dette. Han nævner at man i mange tilfælde kan skalere og approksimere sig til heltal, og at det tidligere var klart hurtigere at regne med heltal. Han ender med at konkludere at den bedste grund til at udvikle eksakte flydende-tals-biblioteker er, at det så vil kunne bruges i mange eksisterende programmer som forventer flydende tal. Derudover nævnes at det i *online*-algoritmer kan være svært og/eller tidskrævende at vedligeholde en uniform skalering, idet data ikke er kendt på forhånd.

En tredie artikel beskriver determinant-udregninger hvor inddata forventes at være heltal, men gemt som et flydende tal (fx i en *double* i C++ med alle decimaler lig nul). Hvis man kun foretager addition, subtraktion og multiplikation med heltal, er der heller intet nøjagtighedsproblem i at gemme heltal i flydende-tals-variabler. Overløb vil altid være et muligt problem, men det problem har også andre løsninger end dem jeg fokuserer på her. Overløb vil dog blive berørt kort i kapitlet om plane sweep-algoritmen.

1.4 Rapportens struktur

I de følgende to kapitler giver jeg en kort beskrivelse af det flydende talsystem og en oversigt over robusthed og metoder til at opnå dette. Derefter følger de tre centrale kapitler der beskriver algoritmerne til løsning af ANN, og hvordan de kan gøres robuste. I kapitel syv beskrives mine implementeringer og de tilhørende testresultater, mens kapitel otte indeholder konklusion og afrunding. Jeg skriver i øvrigt for mig selv, i den forstand at jeg antager læseren mht. datalogisk viden er på det niveau jeg var på ved starten af dette projekt.

1.5 Kildekode

Kildekoden findes her:
`http://www.diku.dk/forskning/performance-engineering/Perfeng/theses.html`.

2

Flydende tal

De flydende tal udgør i en computer en endelig talmængde, som kan kaldes $\mathbb{F}$. For at opnå standardisering introduceredes omkring 1985 IEEE 754[1], *Standard for Binary Floating-Point Arithmetic*. Langt de fleste computere (iflg. [27, s. 17]) implementerer flydende tal iflg. denne standard. Den omhandler bl.a. repræsentation af tal i binært format og principper for beregninger. I det følgende, som er baseret på [21, 27], betyder 'standarden' IEEE 754.

2.1 Formatet

Et flydende talsystem, kaldet et $\mathbb{F}$-system, er defineret ved fire tal:

β = base
p = præcision
e_{min} = mindste eksponent
e_{max} = største eksponent

Et flydende tal, $\tilde{x} \in \mathbb{F}$, består af disse tre dele:

$$\tilde{x} = \underbrace{\pm}_{Fortegn} \underbrace{d_0.d_1d_2...d_{p-1}}_{Signifikand} \times \underbrace{\beta^e}_{Eksponent}$$

I denne sammenhæng er $\beta = 2$. Der findes 4 præcisioner i standarden: *enkelt*, *dobbelt*, *enkelt udvidet* og *dobbelt udvidet*. I tabel 2.1 ses værdier for p, e_{min}, og e_{max}.

	Enkelt	Enkelt udv.	Dobbelt	Dobbelt udv.
p	24	≥ 32	53	≥ 64
e_{max}	$+127$	$\geq +1023$	1023	$\geq +16383$
e_{min}	-126	≤ -1022	-1022	$\leq +16382$
Bits i eksponenten	8	≥ 11	11	≥ 15
Bits i alt	32	≥ 43	64	≥ 79

[1] Kan købes her: http://ieeexplore.ieee.org/servlet/opac?punumber=4610933

At en *double* kan rumme 1 fortegnsbit, 53 signifikand-bits og 11 eksponent-bits i 64 bits skyldes at den mest betydende af de 53 bit i signifikanden ikke gemmes, da den altid er 1. Dette skyldes at tallene gemmes i *normaliseret* form. Et flydende tal i dobbelt præcision er organiseret i computeren på denne måde (mest betydende bits til venstre):

Bit:	63	62-52	51-00
	Fortegn	Eksponent	Signifikand

De 11 bit i eksponenten for en double gemmes som et positivt tal i $[0, 2047]$. For at få den reelle værdi skal der fratrækkes en såkaldt bias på 1023 (127 for enkeltpræcision). Der findes følgende kombinationer (i tabellen er bias fratrukket eksponenten). For alle værdier gælder at de kan være positive eller negative.

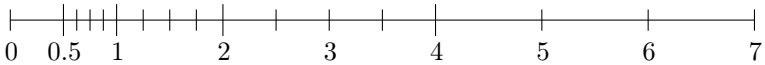
Signifikand	Eksponent		Værdi
0	-1023	$=$	± 0
> 0	-1023	$=$	$\pm$subnormal
$\neq 0$	$[-1022, 1023]$	$=$	$\pm$normaliseret
$\neq 0$	1024	$=$	$\pm\infty$

2.2 Normalisering

Det decimale tal, 0.1, kan repræsenteres som både 1.0×10^{-1} og 0.1×10^{0}. I IEEE 754 er (næsten) alle flydende tal normaliserede, hvilket vil sige at det første ciffer i signifikanden er forskelligt fra 0. Dette gør repræsentationen af et flydende tal unik. Det medfører også, når basen er 2, at man ikke behøver at gemme det første ciffer, og at man derfor får en ekstra bit til signifikanden. I denne form kan man dog ikke repræsentere 0, så derfor kræves en speciel form — alle cifre i signifikanden sættes til 0, og eksponenten til $e_{min} - 1 = -1023$.

Med et endeligt antal cifre er det klart at man ikke kan have samme antal decimaler for meget store tal som for meget små tal. Dette antyder problemerne med operationer på tal af meget forskellig størrelsesorden. Derudover er der også problemer mht. repræsentation af visse decimale tal i binære systemer, fx kan 0.1 decimalt ikke repræsenteres eksakt i et binært system.

Et system med $\beta = 2, p = 3, e_{min} = -1, e_{max} = 2$, går fra 1.00×2^{-1} til 1.11×2^{2}, dvs. decimalt fra 0.5 til 7. Herunder vises talrækken grafisk (kun 0 og positive tal vises):



Hvert repræsenterbart tal i dette $\mathbb{F}$-system er repræsenteret ved en lodret streg. Det er karakteristisk for disse talsystemer, at der dels ingen tal findes tæt ved 0, dels at afstanden mellem to *nabotal* vokser hver gang eksponenten

vokser. Det skyldes normaliseringen; mindste signifikand er 1.00...0 og mindste eksponent er e_{min}, så mindste positive tal er $1.00...0 \times \beta^{e_{min}}$. IEEE 754 giver mulighed for det der kaldes gradvist underløb (eng. *gradual underflow*), hvilket giver flere tal omkring 0. Det opnås ved at tillade ikke-normaliserede tal. I dette eksempel ville man få flg. ekstra tal:

$$0.01_2 \times 2^{-1} = 0.125_{10}$$
$$0.10_2 \times 2^{-1} = 0.250_{10}$$
$$0.11_2 \times 2^{-1} = 0.375_{10}$$

Disse tal, kaldet *subnormale* eller *denormaliserede*, er dog mindre præcise, idet de benytter et ciffer færre end de normaliserede.

I standarden er mindste normaliserede tal i dobbelt præcision 2^{-1022} og det største er $(2 - 2^{-53}) \times 2^{1023}$.

2.3 Afrunding

I IEEE 754 er det standard at afrunde til nærmeste flydende tal med afrunding til lige i tvivlstilfælde. Standarden foreskriver også at der anvendes eksakt afrunding. Eksakt afrunding betyder, at hvis et resultat kan repræsenteres eksakt med p bits, så gemmes det eksakte resultat. Hvis ikke, så gemmes det nærmeste flydende tal der kan repræsenteres i p bit. Hvis man fx i et binært system med 4-bit præcision udfører additionen $1000 + 0.1$ (binære tal), gemmes resultatet 1000. Afrunding til lige betyder at man, i tilfælde af at der er to mulige resultater, vælger det lige tal som resultat. Standarden giver også mulighed for at anvende afrunding mod $\pm\infty$, hvilket bl.a. kan udnyttes i intervalregning.

2.4 Fejlanalyse

Til at tale generelt om fejl og afvigelse benyttes begreberne absolut og relativ fejl (se fx [27]).

Absolut fejl = tilnærmet værdi - eksakt værdi,
Relativ fejl = absolut fejl/eksakt værdi.

Den afvigelse der måtte være mellem et reelt tal, x, og dets flydende-tal-repræsentant, $\tilde{x}$, kan måles på flere måder; absolut fejl, relativ fejl og *ulp* (engelsk for *units in last position*). Den relative fejl er som nævnt $(\tilde{x} - x)/x$. Med ulp angives størrelsen på fejlen: Hvis fx et flydende tal, 3.12×10^{-2} (for $\beta = 10, p = 3$), repræsenterer det reelle tal 0.0314, så er fejlen på to ulp. En ulp kan også ses som størrelsesordenen for den mindst betydende bit i et tal. En ulp står i forhold til et konkret tal, fx er ulp(1) = 0.001, ulp(10) = 0.01 og ulp(1110) = 1, for et binært system med 4-bit præcision.

Som et mål for granulariteten anvendes systemets såkaldte maskinpræcision eller maskinepsilon, ε. Dette tal afhænger af afrundingsreglen:

Afrunding ved afskæring: $\varepsilon = \beta^{1-p}$,
Afrunding til nærmeste: $\varepsilon = \frac{1}{2}\beta^{1-p}$.

I dobbelt præcision og med afrunding til nærmeste er $\varepsilon = 2^{-53}$.

Maskinpræcision er en øvre grænse for fejl:

$$|x - \tilde{x}| \leq \varepsilon |x| \quad \Leftrightarrow \quad |\frac{x - \tilde{x}}{x}| \leq \varepsilon.$$

Det gælder i øvrigt at associativitet ikke er garanteret i beregninger foretaget med flydende tal:
$(1 + \varepsilon) + \varepsilon = 1$, mens $1 + (\varepsilon + \varepsilon) > 1$.

3

Robusthed

Der er muligvis ikke enighed om hvad ordet robust betyder i algoritmisk sammenhæng, men i denne rapport beskæftiger jeg mig kun med metoder der gør algoritmer numerisk eksakte. Det vil altså sige at jeg sætter lighedstegn mellem robuste og eksakte algoritmer, med den tilføjelse at en eksakt algoritme godt kan indeholde både ineksakte og eksakte metoder, blot den garanterer at resultatet er eksakt. Der er tilfælde hvor man er tilfreds med tilnærmede numeriske resultater, med en garanti for at algoritmen stopper eller med at det topologiske element er korrekt. De situationer ser jeg ikke på, men vil her give et overblik over nogle af de eksisterende metoder.

I [40] foreslås en opdeling af algoritmer i to kategorier, der håndterer robusthedsproblemer på forskellige måder:

1 Fejltolerante algoritmer.

2 Eksakte algoritmer.

Algoritmer i den første kategori kan siges at benytte numeriske metoder i minimalt til moderat omfang. De vil kun blive præsenteret i en kort gennemgang i dette kapitel, som er baseret på [40, 46, 47, 49, 35].

3.1 Fejltolerante algoritmer

I denne kategori af algoritmer anvendes flydende tal-beregninger, men det antages at fejl har en begrænset størrelse, dvs. en størrelse man kan leve med. Ideen er at der på trods af fejl kan opnås brugbare resultater. Et eksempel er *ε-geometri* [40]. Heri gøres objekter 'tykkere', så fx et punkt bliver til en disk, og en linie bliver til arealet mellem to parallelle linier (en fed streg). Såkaldte ε-prædikater returnerer et svar der angiver hvor meget (i en eller anden, her uspecificeret, forstand) inddata tilfredsstiller prædikatet. Et andet eksempel er den *aksiomatiske tilgang*, hvor ideen iflg. [40] er, at afdække de egenskaber ved primitive operationer der er essentielle for at bevise en algoritmes korrekthed. Herpå findes så invarianter der udelukkende er baseret på disse egenskaber. Et tredie eksempel er metoder der kaldes *parsimonious*. Det ord kan oversættes til nærig, og det der hentydes til er at man vil undgå at udføre beregninger hvis resultat kan udledes af tidligere beregninger. Et eksempel er Pappus teorem,

hvori en bestemt konfiguration af seks punkter gør at man kan udlede at tre andre punkter ligger på linie.

Blandt metoder med minimal anvendelse af numeriske metoder finder man den topologisk-orienterede, som beskrevet i [47]. Den tager udgangspunkt i to antagelser. A: Logiske og kombinatoriske beregninger kan udføres fejlfrit, mens numeriske beregninger ikke kan. B: Der er som udgangspunkt ingen kendt grænse for de numeriske fejl.

Metoden består, meget kort fortalt, af tre trin. Antag at man skal løse et givet geometrisk problem, P, med en given konventionel algoritme, f. Man danner så en robust algoritme, $\tilde{f}$, på følgende måde:

1. Find de rent topologiske egenskaber, Q, som løsningen skal have, og som kan kontrolleres i rimelig tid. Det er ikke helt klart hvad der menes med rimelig tid, men artiklen nævner at fx egenskaber der er NP-hårde at kontrollere ikke skal med.

2. Beskriv den basale del af algoritmen med topologiske og kombinatoriske termer, på en sådan måde at de fundne egenskaber, Q, garanteres. Dette kaldes at danne det topologiske skelet. Her vil oftest indgå ikke-deterministiske valg. Et eksempel på elementer der kan indgå her er, at opdele en graf i to dele og tilføjelse/sletning af knuder til/fra en graf.

3. Sidste skridt består i, med brug af numeriske metoder, at fastlægge hvordan de ikke-deterministiske dele skal udføres.

Algoritmer dannet med dette princip producerer altid et resultat, men kvaliteten kan variere. I [46] beskrives en topologisk robust implementering af en algoritme der beregner et Voronoi-diagram. I Voronoi-algoritmer benyttes prædikatet incircle, og artiklen beskriver en kørsel af dette program hvor alle kald til deres incircle-prædikat returnerede et tilfældigt tal. Programmet producerede dog stadig et resultat der overholdt de krav der opstillet. Jeg har ikke brugt meget tid på denne metode, og det er ikke klart for mig hvordan det skal gøres i praksis, men det virker interessant at fokusere på topologiske egenskaber.

3.2 Eksakte algoritmer

Som sagt er en eksakt algoritme her en der garanterer et korrekt resultat. Det kan ske med en udelukkende eksakt beregning, men det kan også ske med en blanding af ineksakte og eksakte beregninger. Det første kaldes ofte multipræcisionsmetoder, mens det andet ofte kaldes filtermetoder. I et filter benyttes flydende tal-beregning kombineret med fejlanalyse. Hvis analysen siger god for beregningen accepteres den, ellers benyttes en bedre metode, og det kan i sidste ende betyde multipræcisionsmetoder.

Valget af den konkrete tilgang afhænger af flere ting, bl.a. af problemets natur, hastighedskrav og selvfølgelig af evt. krav til nøjagtighed af resultatet. Flere artikler kategoriserer algoritmer efter kvalitet. Iflg. [43] kan en geometrisk algoritme være *eksakt*, dvs. altid korrekt, *robust*, dvs. altid korrekt for en eller anden perturbation af inddata, *stabil*, dvs. hvor perturbation er lille, og ned til *skrøbelig* som ikke garanterer at levere et brugbart resultat.

3.2.1 Metoder

Som nævnt kan man skelne mellem multipræcisions- og filtermetoder. De første går ud på at sikre sig at der altid er tilstrækkelig præcision til at undgå afrundingsfejl, mens de andre først forsøger med en unøjagtig metode, og med fejlanalyse bagefter undersøger om resultatet er troværdigt.

Multipræcision

Hvis man vil opnå (principielt) uendelig præcision må man gå fra at benytte de hurtige hardware-baserede beregninger til delvist at benytte software. Denne tilgang medfører næsten altid langsomme beregninger. Eksempler er GMP, LEDA og CGAL — se hhv. [20, 2, 11]. Alle de tre nævnte biblioteker har både heltals- og flydende tals-versioner af multipræcisionstyper. Det er i en konkret situation ofte muligt at begrænse sig til blot en udvidet, og ikke uendelig, præcision. Dette kræver at størrelsen af inddata har en kendt øvre grænse. Derudover skal man udregne hvor store udtrykkene kan blive, hvorved man får en øvre grænse for præcision. Det er dog ikke oplagt hvordan man udnytter dette. Jeg er ikke stødt på et multipræcisions-bibliotek der tillader at sætte en grænse for præcision. I praksis vil man derfor benytte et af de nævnte biblioteker, som netop håndterer uendeligt store størrelser.

For alle disse typer gælder at de benytter en struktur der tillader et tal at vokse. I LEDAs integer-type er et tal gemt som en vektor af *unsigned long*-typer, med fortegn og størrelse gemt i ekstra variabler. I [33] findes algoritmer til addition, subtraktion, multiplikation og division af sådanne store tal.

Filtermetoder

I mange tilfælde kan resultatet af en udregning med flydende tal godt bruges, og det udnyttes i filtermetoder. Der findes forskellige former for filtre, og de opdeles typisk i tre grupper:

1. Statiske, hvor fejlgrænsen udregnes en gang for alle. Det kræver en kendt øvre grænse for den absolutte værdi af inddata.
2. Semi-statiske, hvor fejlgrænsen udregnes undervejs, men dele af denne udregning dannes en gang for alle.
3. Dynamiske, hvor ingen udregning udføres før algoritmen går i gang.

Filtre kan kobles sammen, fx kan man i samme program først bruge et semi-statisk, fulgt af et dynamisk hvis det første fejler, og eksakt beregning hvis det dynamiske fejler.

Et eksempel på et semi-statisk filter er i bestemmelse af fortegnet for et udtryk. Her kan man nøjes med at sikre sig at resultatet er tilstrækkeligt lang fra 0: Som det sås tidligere er $\varepsilon\tilde{x}$ den maksimale absolutte fejl i udtrykket $\tilde{x}$, beregnet med flydende tal. Hvis $|\tilde{x} - \varepsilon\tilde{x}| > 0$ kan man stole på fortegnet af $\tilde{x}$.

Intervalregning

Som antydet herover er man tit kun interesseret i at et resultat ligger i et givet interval, og ikke den eksakte værdi. Intervalregning er en metode hvori tal ses som intervaller. Et sådant interval består af to flydende tal, og beregninger foregår på disse intervaller. Et tal repræsenteres ved et endeligt interval: $[x] = [\underline{x}, \overline{x}]$. Operationer på disse er defineret som følger:

$$[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$[x] - [y] = [\underline{x} - \underline{y}, \overline{x} - \overline{y}]$$
$$[x] \cdot [y] = [min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}]\}, max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}]\}$$

Hvis man udregner et resultat med intervalregning, og kun søger fortegnet, kan man stole på resultatet hvis intervallet ikke indeholder 0. I [9] beskrives hvordan man udnytter IEEE 754 i denne forbindelse. For at sikre mod afrundingsfejl rundes den nedre grænse ned til det nærmeste mindre flydende tal, og den øvre grænse rundes op til det nærmeste større flydende tal. IEEE 754 giver mulighed for afrunding mod $\pm\infty$. Det kræver dog at processoren sættes i den rette tilstand, så for at lette beregningerne gemmes et interval ikke som $[x] = [\underline{x}, \overline{x}]$, men som $[x] = [-\underline{x}, \overline{x}]$. Det gør det muligt altid at bruge afrunding mod ∞, man skal altså ikke ændre afrundingsprincip imellem beregninger.

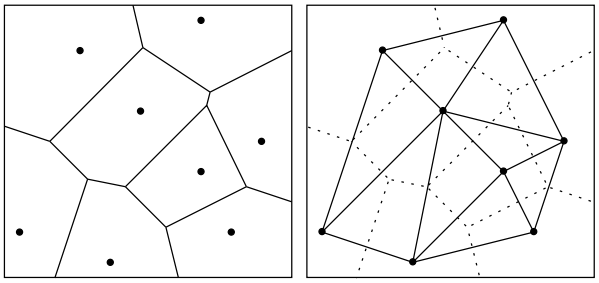
Perturbering af data

Mange problemer i forbindelse med beregning af determinanter (i geometrisk sammenhæng) opstår når data ikke er i den såkaldte generelle position. Der argumenteres i nogle artikler for at man perturberer data, så de kommer i generel position. Det er relevant i de tilfælde hvor data er målte størrelser, og derfor udsat for en vis usikkerhed, fx aminosyrer i et protein. I den forbindelse kan det nævnes at konvertering fra decimal til binær form også indeholder risici. Disse emner kommer jeg dog ikke nærmere ind på i denne rapport, men [40, 49] indeholder mere information.

4

Triangulering

En triangulering af en punktmængde i planen danner en planar graf. I en sådan graf er der (langt) færre kanter end i en komplet graf, hvilket gør det (langt) hurtigere at finde den nærmeste nabo for et punkt (via punktets 'forbundne' naboer, dvs. kanterne der udgår fra det) end med brute-force. Derfor kaldes triangulering i denne sammenhæng af og til for en filtrering, idet man filtrerer irrelevante data bort, og derpå løser opgaven for en mindre datamængde. I dette kapitel beskriver jeg Delaunay-triangulering, men starter dog med en nært beslægtet struktur. Et *Voronoi-diagram* for punktmængden $S \subset \mathbb{R}^2$ indeholdende n punkter kaldes $\mathrm{Vor}(S)$, se fig. 4.1.



Figur 4.1: *Til venstre et Voronoi-diagram for en punktmængde. Til højre en Delaunay-graf for samme punktmængde.*

$\mathrm{Vor}(S)$ inddeler planen i n regioner. Hver region er tilknyttet et punkt i S, dette punkt kaldes *generatoren* for regionen. Kanterne der markerer regionerne kaldes *Voronoi-kanter*. En region afgrænser det område af planen hvori man er tættere på regionens generator end på nogen andet punkt i S. Skæringspunkter mellem kanter kaldes *Voronoi-punkter*. To generatorer, hvis regioner deler en Voronoi-kant, kaldes *Voronoi-naboer*. Principielt kan man, med de rette datastrukturer og mulighed for at traversere disse, let finde en generators nærmeste nabo-generator. Det kræver blot at man for hver region har adgang til dens Voronoi-kanter, og for hver kant har adgang til dens to regioner samt sidstnævntes generator — se [6].

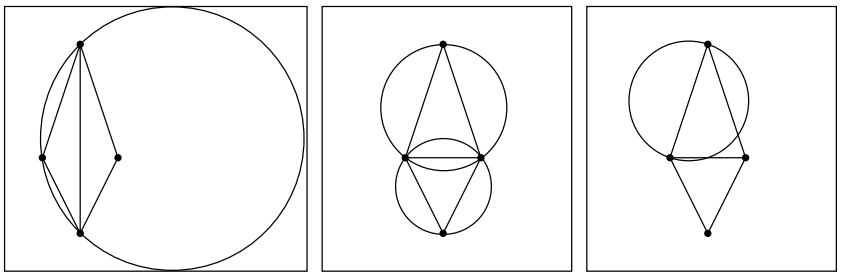
$\mathrm{Vor}(S)$ har en dual graf, kaldet en *Delaunay-graf* eller $\mathrm{DG}(S)$. Den har et

punkt eller knude[1] for hver generator i Vor(S), en facet for hvert Voronoi-punkt, og der er en kant mellem to knuder i DG(S) hvis de to tilsvarende generatorer i Vor(S) har en fælles kant, dvs. de er Voronoi-naboer — se [6]. Med Delaunay-grafen er det, i hvert fald konceptuelt, let at løse ANN. Man skal blot for hver knude traversere dennes naboknuder, udregne afstanden til hver af dem og gemme den nærmeste.

En Delaunay-graf for punktmængden S besidder bl.a. følgende to egenskaber [6, s. 190] — se fig. 4.2:

D1: Tre punkter $p, q, r \in S$ er knuder i samme facet i Delaunay-grafen DG(S) hvis og kun hvis cirklen gennem de tre punkter ikke indeholder noget punkt fra S.

D2: To punkter $p, q \in S$ danner en kant i DG(S) hvis og kun hvis der findes en cirkel gennem p og q som ikke indeholdet noget punkt fra S.



Figur 4.2: *Til venstre en lovlig triangulering som ikke er en lovlig Delaunay-graf (overtræder det første af de to krav herover). I midten og til højre lovlige Delaunay-grafer.*

En subdivision kan defineres som en planar *embedding* af en graf, udelukkende med lige liniestykker [6]. Man kan så definere en triangulering af en punktmængde $S \subset \mathbb{R}^2$ som en maksimal planar subdivision, sådan at man ikke kan tilføje en kant uden at ødelægge planariteten. Dette medfører bl.a. at kanten af trianguleringen udgør en konveks polygon indeholdende alle punkter i S.

Hvis en punktmængde S er i såkaldt *generel position*, vil det i denne sammenhæng sige at S ikke indeholder fire punkter der ligger på samme cirkel. Hvis S er i generel position, udgør Delaunay-grafen DG(S) automatisk en triangulering af S. Hvis S ikke er i generel position, vil man skulle tilføje kanter til DG(S) for at gøre den til en triangulering. For at kunne tilføje en kant, må man kunne afgøre om en eller flere kanter derved bliver ulovlige, dvs. overtræder D1 eller D2. Det gøres med brug af incircle-funktionen.

Dannelse af en Delaunay-graf, DG(S), for n punkter i planen tager $O(n \log n)$ tid ([25, 6, 15]) mens ANN-problemet ved hjælp af DG(S) kan løses i $O(n)$ tid, da summen af alle knuders valenser i DG(S) er lineær i antal knuder, som bevist

[1] Disse to ord anvendes synonymt i dette afsnit, med mindre andet fremgår.

i det følgende.

Lemma 1 (fra [6]). Lad S være en punktmængde med n punkter i planen, hvoraf ikke alle er kollineære. Antag at k af de n punkter ligger på kanten af det konvekse hylster for S. Enhver triangulering af S har i alt $3n - k - 3$ kanter.

Bevis. Lad T være en triangulering af S. Antag at T består af m trekanter og $m+1$ facetter (hvor den ene er den åbne, omgivende facet). Hver trekant har tre kanter, mens den åbne facet har k kanter. Hver kant i T, som har maksimalt et af sine endepunkter på kanten af det konvekse hylster, støder op til to trekanter, og alle kanter på kanten af det konvekse hylster støder op til en trekant og den åbne facet; hver kant støder altså op til to facetter, og det samlede antal kanter er derfor $(3m + k)/2$.

Lad $G = (V, E)$ være en forbundet planar graf, bestående af knudemængden V og kantmængden E. Antag at G består af n_G knuder, e_G kanter og f_G facetter. For en sådan graf gælder iflg. Euler: $e_G = n_G + f_G - 2$.

Indsættes værdierne foroven i Eulers formel, fås følgende:

$$
\begin{aligned}
&(3m + k)/2 = n + m + 1 - 2 \Leftrightarrow \\
&m = (3m + k)/2 - n + 1 \Leftrightarrow \\
&2m = 3m + k - 2n + 2 \Leftrightarrow \\
&m = 2n - k - 2 \Rightarrow \\
&(3m + k)/2 = (3(2n - k - 2) + k)/2 \Leftrightarrow \\
&(3m + k)/2 = 3n - k - 3.
\end{aligned}
$$

□

4.1 Algoritmen

Der findes forskellige typer af trianguleringsalgoritmer, bl.a. del og hersk [25], *plane sweep* [16] og en inkrementel (evt. randomiseret) version [24]. Jeg har valgt to eksisterende biblioteker som grundlag for eksperimenter: Triangulation Template Library (TTL) [29] og Computational Geometry Algorithms Library (CGAL) [11]. Begge benytter en inkrementel algoritme til triangulering. TTL blev mere eller mindre tilfældigt foreslået, men fordi det bl.a. er simpelt og lettilgængeligt mente jeg at det var relevant at undersøge robustheden, specielt med henblik på at forbedre denne ved brug af eksterne metoder. CGAL er et kendt bibliotek, ikke mindst mht. robusthed, og det var derfor oplagt at se hvordan det fungerer i praksis. Fokus er her på CGALs egne metoder, ikke på at tilføre robusthed med eksterne metoder. Andre muligheder er *triangle* [42] og *Qhull* [39]. Alle de nævnte biblioteker er åbent software, i det mindste i den forstand at kildekoden er tilgængelig.

I både TTL og CGAL anvendes *iteratorer*. En iterator er en form for peger, og giver adgang til elementer i en datastruktur. De følgende tre koncepter er centrale [45, s. 550]:

- det element der peges på

- flyt til det næste/forrige element
- lighed

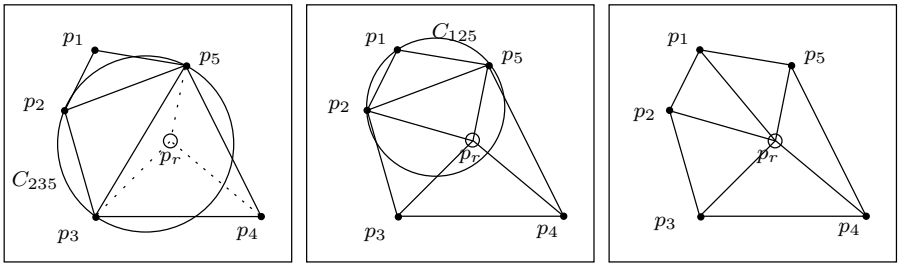
Iteratorer tænkes anvendt på lineære strukturer som fx lister. De har alle et start- og et slutelement. Der findes også cirkulære strukturer, fx alle kanter der udgår fra en given knude i en graf. I CGAL og til dels TTL opererer man derfor med *cirkulatorer*, der kan beskrives som iteratorer for cirkulære strukturer.

Den inkrementelle algoritme foretager Delaunay-triangulering af en punktmængde $S \subset \mathbb{R}^2$ med disse trin:

INKREMENTEL-DELAUNAY-TRIANGULERING(S)
1 Dan initiel triangulering T, bestående af en/to trekanter der omslutter S
2 **for** ($r \leftarrow 1$ **to** n)
3 Lokaliser den trekant $t \in T$ der indeholder p_r
4 Dan nye kanter fra p_r til hvert punkt i t
5 Lovliggør de tre oprindelige kanter i t.
6 Slet omsluttende trekant(er) fra T.

Der er flere metoder til punktlokalisering af en nyt punkt p_r i en triangulering T:

1. *vandring (walk)* Man starter et tilfældigt sted og vandrer ad eksisterende kanter i T til den rette trekant. Iflg. [25] beskrevet i [24].
2. *hop+vandring (jump and walk)* Her vedligeholdes en struktur med et (mindre) antal punkter, kaldet *ankre*, i T. Man finder sekventielt ankret nærmest p_r og derfra vandres til den rette trekant. Se [37].
3. Forskellige metoder der benytter mere avancerede datastrukturer som fx orienterede acykliske grafer [6] og en metode kaldet *Delaunay-hierarkier* [14].



Figur 4.3: *Til venstre en triangulering efter p_r er tilføjet, men inden lovliggørelse. I midten efter lovliggørelse af kant p_3p_5. Til højre efter lovliggørelse af kant p_2p_5.*

Lovliggørelse sker ved brug af incircle-testen. Idet et nyt punkt p_r indsættes i trekant $t \in T$ dannes nye kanter fra p_r til hver af de tre knuder i t. De nye kanter er lovlige allerede ved indsættelsen, men andre skal undersøges. Denne proces foregår rekursivt og starter med de tre oprindelige kanter i t — se fig.

4.3. Lovliggørelse af T efter indsættelse af punkt p_r indebærer altid at det er punktet p_r der undersøges for om det er inde i, uden for eller på cirklen gennem tre andre punkter.

At de nye kanter der dannes ved indsættelse af et nyt punkt er lovlige ses af følgende argument. I fig. 4.3 til venstre kan cirklen C_{235} gennem p_2, p_3 og p_5 formindskes så den går gennem p_3 og p_r og er tom. Altså vil den nye kant opfylde krav D2. Samme argument kan anvendes på de to andre nye kanter. Et tilsvarende argument bruges til at vise at en vendt kant er lovlig. I fig. 4.3 til venstre er kant p_3p_5 ulovlig så den vendes og bliver til kanten p_2p_r (figuren i midten). Cirklen C_{125} kan også her formindskes så den går gennem p_2 og p_r og er tom.

At ingen kanter undersøges mere end en gang for hvert nyt punkt kan vises på følgende måde. Ved indsættelse af et nyt punkt p_r gælder at en kant k skal testes hvis og kun hvis der er netop en kant k' imellem k og p_r, og k' er blevet vendt (eller dannet, som de tre nye kanter). I fig. 4.3 til venstre ses at kanten p_3p_5 er ulovlig. Den vendes og bliver til kant p_2p_r (figuren i midten). De næste to kandidater til lovliggørelse er kanterne p_2p_3 og p_2p_5. I fig. 4.3 til højre er kant p_2p_5 vendt til kanten p_1p_r. Da kanter der vendes altid får det nye punkt, p_r, som ene endepunkt kan en kant der er vendt altså ikke blive vendt igen, under indsættelse af samme punkt. Algoritmen stopper altså.

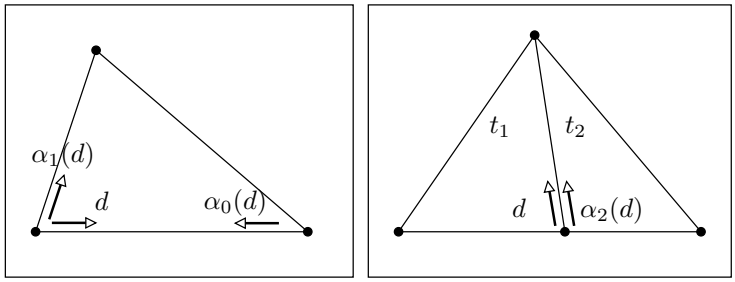
Hvis et nyt punkt p_r ligger på en eksisterende kant, gøres i [6] det at man sletter den kant punktet findes på, og der dannes fire nye kanter til hjørnerne i den firkant der opstår. I TTL sker det samme som ovenfor, dvs. først dannes kanter fra p_r til de tre knuder i den lokaliserede trekant. To af disse nye kanter vil være sammenfaldende med den kant i den oprindelige trekant som det nye punkt ligger på. Denne oprindelige kant vil blive vendt.

4.2 Triangulering med TTL

TTL er et bibliotek udelukkende til Delaunay-triangulering. Det er baseret på generisk programmering (se fx [45]), hvilket medfører at brugeren af biblioteket i princippet selv kan tilføre visse dele af et endeligt program. Det drejer sig typisk om datastrukturer og visse operationer på disse. Der er dog mht. TTL lavet eksempelkode, så der findes et fungerende program til Delaunay-triangulering, hvilket jeg benytter mig af.

TTL bruger en såkaldt *halvkant* (eng.: *half-edge* - se [6]) til at implementere trianguleringen. En halvkant udgør en side af en trekant, den har en retning, en tilknyttet knude, en tvillingekant (medmindre det er en yderkant) og en efterfølger. En halvkant tilhører altid netop en trekant. Hver trekant repræsenteres ved en primærkant. Denne har en liste med de to andre kanter tilknyttet, så den samlede triangulering består af en liste af primærkanter. Til navigation benyttes et begreb kaldet en *dart*. En dart, $d = (v_i, e_j, t_k)$ (se fig. 4.4), defineres af tre elementer: v_i er knuden man står i, e_j er kanten man er på og t_k er trekanten man er i. En dart har også en retning, den peger fra sin kildeknude mod den modstående knude. I udgangspunktet er retningen altid mod uret mht. den

trekant den er i, se fig. 4.4.



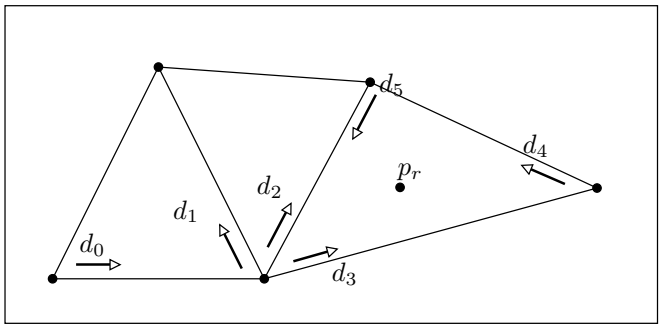
Figur 4.4: *Darts og tilhørende operationer $\alpha_0, \alpha_1, \alpha_2$, på dem.*

Der er tre operationer på en dart d: $\alpha_0(d)$ skifter knude og fastholder kant og trekant. $\alpha_1(d)$ skifter kant, og fastholder knude og trekant. $\alpha_2(d)$ skifter trekant og fastholder knude og kant. Bemærk at hver af disse ændrer orientering for darten. Det gælder ydermere at $d = \alpha_n(\alpha_n(d))$ for $n \in (0, 1, 2)$, og hvis d er på yderkanten af en triangulering, er $\alpha_2(d) = d$. Ved komposition fås det der kaldes en *n-orbit*; fx er en 0-orbit defineret ved $\alpha_2(\alpha_1(d))$. Gentagen applikation af en 0-orbit vil besøge en knudes samtlige kanter.

Der er i [29] ingen forklaring på hvorfor man indledningsvis danner to omsluttende trekanter; i [25, 26, 6] dannes kun en, men da man først omslutter S med to trekanter vil ethvert punkt i S ligge i en trekant, så punktlokalisering finder altid en trekant. TTL anvender darts på flg. måde til at finde denne trekant.

4.2.1 Punktlokalisering

Det nævnes i [29] ikke hvor lokaliseringsalgoritmen stammer fra, men den minder om vandre-metoden. Jeg antager det er derfor inddata i TTL sorteres. Når punkt p_{r+1} indsættes har man en dart fra indsættelse af punkt p_r. Hvis data er jævnt fordelt vil darten fra forrige punktindsættelse altså være relativt tæt på trekanten hvori næste punkt ligger.



Figur 4.5: *Punktlokalisering med darts. Udgangspunktet er d_0.*

Givet to darts d og $\alpha_0(d)$ og et punkt p, kan man med funktionen orient2d undersøge hvor p ligger i forhold til linien gennem de to punkter repræsenteret

ved d og $\alpha_0(d)$. Hvis p ligger til venstre for alle tre kanter i en trekant t kan man konkludere at punktet ligger i denne trekant. Hvis p ligger til højre for en af kanterne, ved man at man skal skifte trekant. Se fig. 4.5.

4.2.2 Lovliggørelse af kanter

Dette sker som nævnt ovenfor med brug af incircle-testen.

4.2.3 ANN med TTL

For at kunne løse ANN med TTL kræves kun at man tilføjer en simpel funktion, som antydet tidligere: Der dannes en Delaunay-triangulering, T, for punkterne i S, og derpå traverseres alle knuder i T. For hver knude traverseres alle dens kanter, den kvadrerede afstand til kantens anden knude udregnes og den nærmeste knude gemmes.

4.2.4 Kompleksitet

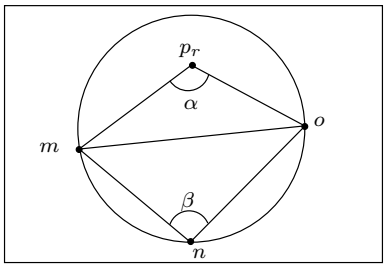
Inkrementel Delaunay-triangulering har en værste-falds tidskompleksitet på $O(n^2)$, men forventet tidskompleksitet på $O(n \log n)$ [14, 6] for den randomiserede udgave. Den kvadratiske tid skyldes at punkterne kan ligge på en sådan måde at man ved lokalisering skal igennem alle eksisterende trekanter for hvert nyt punkt. Det samme gælder for antal kanter der skal lovliggøres - her gives ikke garanti for andet end at algoritmen maksimalt skal undersøge alle eksisterende kanter for hvert punkt der indsættes. I [29] står intet specifikt om kompleksitet for punktlokalisering, men metoden er formentlig ikke dårlig for punkter der ligger jævnt fordelt og relativt tæt på hinanden. I [25] nævnes at vandre-metoden for sådanne inddata muligvis kun bruger konstant tid. Dette vil også gælde her, som omtalt tidligere.

4.2.5 Algoritmisk robusthed

Hvis man giver TTL-programmet inddata bestående af fire kollineære punkter går det ned. Jeg kan se at det sker i forbindelse med sletning af de to omsluttende trekanter, men har ikke haft tid til at gå dybere ind i det. Det var muligvis en ide at undersøge om behandling af punktindsættelse på en eksisterende kant skulle ændres, så algoritmen følger andre versioner, fx [6].

4.2.6 Numerisk robusthed

Der er i TTL-implementeringen mulige robusthedsproblemer i forbindelse med to opgaver, punktlokalisering og kantlovliggørelse. Punktlokalisering udføres som sagt ved brug af orient2d-funktionen, og lovliggørelse af kanter løses ved brug af incircle-funktionen. Med hensyn til punktlokalisering skriver forfatterne selv at deres metode er *a finite arithmetic fast version*, og at den kan gøres mere robust med metoderne i [41]. Til kantlovliggørelse benyttes en metode der iflg. [29] stammer fra [12], og førstnævnte siger at metoden iflg. [12] er numerisk stabil. Metoden går blot ud på at undersøge de to modstående vinkler i den konvekse firkant der dannes af fire punkter — i fig. 4.6 α og β. Hvis $\alpha + \beta > \pi$



Figur 4.6: *Lovliggørelse af kant. Hvis p_r ligger inde i cirklen gennem de tre andre punkter, skal den eksisterende kant fjernes. Det kan afgøres hvis man kender de to vinkler, α og β.*

skal kanten mo erstattes af kanten $p_r n$. Med rationelle tal kan vinklerne udregnes eksakt ud fra de fire punkter: $\cos(\alpha) = (\overrightarrow{p_r m} \cdot \overrightarrow{p_r o})/|\overrightarrow{p_r m}| \cdot |\overrightarrow{p_r o}|$

I selve TTL-koden, der ikke benytter rationelle tal, gøres der opmærksom på at der eksperimentelt er fundet eksempler på numerisk ustabilitet i den ovenfor beskrevne metode. Der er derfor indbygget et ekstra check, som dog tilsyneladende kun benyttes ved *constrained* Delaunay-triangulering. I [29] nævnes også at der er indbygget en mekanisme til at undgå cykler og uendelige løkker i tilfælde af at fire eller flere punkter ligger på eller næsten på samme cirkel. De er altså opmærksomme på at metoden ikke er eksakt.

Da TTL er tænkt som et *template*-bibliotek er visse dele gjort udskiftelige ved at udskille og placere dem i separate klasser, som så anvendes i den generiske triangulering. Man[2] kan sige at den opgave programmet løser har særlige karaktertræk, og derfor kaldes en sådan klasse for *træk* (eng.: *traits*) for triangulеringen. De to interessante dele er her prædikaterne orient2d og incircle. Det er lykkedes at gøre orient2d-funktionen let udskiftelig, idet den ligger i traits-klassen. For incircle-funktionen er det lidt anderledes, idet den overordnede metode, dvs. den udviklet af [12], er placeret i selve biblioteket, mens udregning af de trigonometriske værdier (som metoden benytter) findes i traits-klassen.

Jeg forsøger at sikre robusthed i TTL med to forskellige filtermetoder. Den første, beskrevet i [41], benytter flydende tal, fejlanalyse og adaptiv beregning. Den anden, beskrevet i [3], udnytter lineær algebra. I [29] foreslår forfatterne selv at robusthed kan forbedres ved at benytte metoderne beskrevet i [41].

Konkret har jeg gjort det ved at lave TTL-biblioteket om til en klasse hvor de relevante dele, nemlig darts, punkter og prædikater, er template-parametre. Jeg har derfor tre forskellige traits-klasser, en med de originale metoder, en med [41]'s metoder og en med [3]'s metoder. Ved at placere prædikaterne i separate klasser og lade dem være parametre i biblioteket, er det meget simpelt at skifte mellem robust og ikke-robust opførsel.

[2] Jeg ved ikke hvor begreberne *trait* og *policy* stammer fra, men de kan findes beskrevet bl.a. i [45] og i [48]

Filtermetode 1

Metoderne i [41] er en videreudvikling af andre tilsvarende metoder, bl.a. med den ændring at de kun retter sig mod binære computere. Dette gør dem hurtigere end de tidligere. Metoderne er beregnet på flydende tal som inddata og stiller krav til både inddata og den computer det skal køre på. For at undgå over/underløb skal inddata til de anvendte funktioner have eksponenter i intervallet $[-142, 201]$. Med hensyn til selve computeren skal aritmetikken være binær, for det andet skal den benytte eksakt afrunding (med afrunding til lige i tvivlstilfælde), og for det tredie skal afrunding ske i dobbelt præcision. Det første er formentlig ikke noget problem, mens det andet følger hvis ens hardware lever op til IEEE 754. Det tredie krav er opfyldt uden videre for nogle computere, men Intel benytter (i Pentium etc. i det mindste) interne registre med udvidet præcision. Det betyder at man i visse operationer har større præcision end normalt. Det er normalt en god ting, men altså ikke med disse eksakte metoder. For at sikre afrunding i dobbelt præcision skal man derfor i visse tilfælde gøre noget aktivt selv. Dette kan være i form af parametre til compileren eller udførsel af en maskininstruktion ved programmets start.

Overordnet består metoden af tre dele: basale eksakte regnealgoritmer, geometriske prædikater (orient2d og incircle) og fejlanalyse. Hvert prædikat har fire trin, som hver danner et resultat: A, B, C, D. A er dannet kun ved brug af flydende-tal-operationer, mens B og C har større præcision og D er det eksakte resultat. Brugeren af prædikatet (dvs. programmøren) ser kun en funktion, orient2d, som selv finder ud af hvornår der er nok viden til at returnere et korrekt svar. Fejlanalyse anvendes til at afgøre om resultatet af et trins udregning er tilstrækkeligt. Hvis ikke, så udføres næste trin indtil det sidste som leverer et eksakt resultat. En vigtig detalje er at et resultat dannet i et trin genbruges i næste, hvis næste trin bliver nødvendigt.

De basale algoritmer består af addition, subtraktion og multiplikation, både af flydende tal og af såkaldte *ekspansioner*. For at undgå overløb gemmes et tal x som ekspansioner, $x = x_n + ... + x_2 + x_1$. De enkelte elementer er ordnet i faldende orden, og skal yderligere være *ikke-overlappende*. Det sidste vil for to tal x og y sige, at den mindst betydende bit (som ikke er nul) i x er større end den mest betydende bit (som ikke er nul) i y, eller omvendt; fx overlapper 100 og 10 ikke, mens 101 og 1 gør. Hvert element i en ekspansion opbevares i en selvstændig *double*. De eksakte metoder regner altså med disse tal, og udregner også fejlen ved en flydende-tal-operation.

Til brug i prædikaterne udregnes en gang for alle fejlgrænser i starten af programmets levetid. Det følgende eksempel er fra [41], med enkelte tilføjelser af mig, og viser hvordan man finder fejlgrænsen for den første værdi, A, for orient2d. Inddata til orient2d er tre punkter, a, b, c (jeg bruger a, b, c her fordi det originale eksempel gør det). Det man søger er fortegnet for denne determinant:

$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} = (a_x - c_x)(b_y - c_y) - (a_y - c_y)(b_x - c_x)$$

I det følgende udtrykker $+, -$ og almindelig multiplikation som fx ab de

eksakte regneformer, mens $\oplus, \ominus$ og $\otimes$ er de tilnærmede flydende tal-operationer. Iflg. [41] er den nemmeste måde at fejlanalysere et udtryk beregnet i flydende tal på, at udtrykke den eksakte værdi som den beregnede plus en ukendt fejl, hvis størrelse er begrænset. For et udtryk $a \oplus b$ er fejlen ikke større end $\varepsilon|a \oplus b|$. Derudover er fejlen også mindre end $\varepsilon|a + b|$. Lad i det følgende t_i angive den eksakte værdi mens x_i angiver den tilnærmede værdi.

$$\begin{array}{ll} t_1 = a_x - c_x & x_1 = a_x \ominus c_x \\ t_2 = b_y - c_y & x_2 = b_y \ominus c_y \\ t_3 = a_y - c_y & x_3 = a_y \ominus c_y \\ t_4 = b_x - c_x & x_4 = b_x \ominus c_x \\ t_5 = t_1 t_2 & x_5 = x_1 \otimes x_2 \\ t_6 = t_3 t_4 & x_6 = x_3 \otimes x_4 \\ t_A = t_5 - t_5 & A = x_5 \ominus x_6 \end{array}$$

A er altså resultatet af den første rutine i orient2d, og udregnet udelukkende med flydende tal-operationer. Af ovenstående ses at $t_1 = x_1 \pm \varepsilon|x_1|$, og tilsvarende for t_2, t_3, t_4. Heraf fås følgende:

$$\begin{array}{rcl} t_5 = t_1 t_2 & = & (x_1 \pm \varepsilon|x_1|)(x_2 \pm \varepsilon|x_2|) \\ & = & x_1 x_2 \pm (x_1 \varepsilon|x_2| + x_2 \varepsilon|x_1| + \varepsilon^2|x_1||x_2|) \\ & = & x_1 x_2 \pm (2\varepsilon + \varepsilon^2)|x_1 x_2| \end{array}$$

$x_5 = x_1 \otimes x_2 = x_1 x_2 \pm \varepsilon|x_5| \Rightarrow x_1 x_2 = x_5 \pm \varepsilon|x_5|$. Dette medfører følgende:

$$\begin{array}{rcl} t_5 & = & x_5 \pm \varepsilon|x_5| \pm (2\varepsilon + \varepsilon^2)(|x_5| \pm \varepsilon|x_5|) \\ & = & x_5 \pm \varepsilon|x_5| \pm (2\varepsilon|x_5| + 2\varepsilon^2|x_5| + \varepsilon^2|x_5| + \varepsilon^3|x_5|) \\ & = & x_5 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_5| \end{array}$$

Tilsvarende gælder for t_6. Nu kan man udregne et udtryk for t_A, den eksakte værdi for A:

$$\begin{array}{rcl} t_A = t_5 - t_6 & = & x_5 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_5| - (x_6 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_6|) \\ & = & x_5 - x_6 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_5| \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)|x_6| \\ & = & x_5 - x_6 \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|) \\ & = & A \pm \varepsilon|A| \pm (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|) \end{array}$$

Kun fortegnet for A er interessant, og det kendes med sikkerhed hvis flg. gælder, dvs. hvis A er så tæt på nul som fejlgrænsen tillader og stadig større end højresiden:

$$(1 - \varepsilon)|A| > (3\varepsilon + 3\varepsilon^2 + \varepsilon^3)(|x_5| + |x_6|)$$

Det er sandt hvis følgende holder:

$$|A| \geq (3\varepsilon + 6\varepsilon^2 + 8\varepsilon^3)(|x_5| + |x_6|)$$

Her fås dog afrundingsfejl, så der skal ganges med $(1+\varepsilon)^2$ (grænsen hæves med fejlen på de to faktorer): $(1+\varepsilon)$ for addition af x_5 og x_6, og $(1+\varepsilon)$ for multiplikationen. Hvis følgende holder kan man stole på A's fortegn:

$$|A| \geq (3\varepsilon + 12\varepsilon^2 + 24\varepsilon^3) \otimes (|x_5| \oplus |x_6|)$$

Den første faktor kan dog ikke udtrykkes i p bit, så der skal rundes op. Den endelige koefficient bliver derfor:

$$3\varepsilon + 16\varepsilon^2$$

Dette tal udregnes ved start af programmet og genbruges derefter ved hvert kald af orient2d. Der er tilsvarende forudberegnede koefficienter for B og C.

I [41] vises resultatet af eksperimenter med metoderne i en Delaunay-triangulering i både to og tre dimensioner. Der benyttes tre typer af inddata:

1 Tilfældige punkter, ligefordelt.

2 Punkter omtrentligt på kanten af en cirkel.

3 Punkter på et kvadratisk *grid* roteret så det ikke følger koordinatakserne.

I hver kørsel er der 1.000.000 punkter. Nedenstående tabel gengiver et udsnit af resultaterne for triangulering i to dimensioner. Der er bemærkelsesværdigt få kald til andre end den første og hurtige version A:

	Tilfældige ligefordelte	Omtrentligt på cirkelkant	Roteret grid
orient2d-kald:			
A	9.497.314	6.291.742	9.318.610
B	0	0	121.081
C	0	0	118
D	0	0	3
incircle-kald:			
A	7.596.885	3.970.796	7.201.317
B	0	50.551	176.470
C	0	120	47
D	0	0	4
Køretider, sekunder:			
Ikke-robust (dvs. A):	57.3	59.9	48.3
Robust (dvs. A-D):	61.7	64.7	62.2

Implementeringen var oprindeligt programmeret i C, men jeg flyttede den over i en C++-klasse. Da min computer er udstyret med en AMD-chip antager jeg at den bruger extended precision, og derfor kalder jeg i konstruktoren denne funktion [23], med argumentet `0x27F`, for at fremtvinge afrunding i dobbelt præcision:

```
void setFpu(unsigned int mode) { asm("fldcw %0": : "m"(*& mode)); }
```

Filtermetode 2

Metoden beskrevet i [3] består i udregning af determinanter for 2×2 og 3×3 matricer, og bruges i hhv. orient2d- og incircleprædikaterne. Det kræves at inddata er heltal med max. 53 hhv. 51 bit præcision. Jeg bruger begge prædikater, så jeg kræver for nemheds skyld at præcision aldrig er større end 51. Der findes en implementering af prædikatet i [1].

2×2-matrix.
I dette tilfælde søger man fortegnet for denne determinant:

$$\mathbf{D} = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = x_1 y_2 - y_1 x_2.$$

I en række tilfælde er det nemt at finde fortegnet for $\mathbf{D}$, fx hvis et ulige antal elementer er negative. Antag at $x_1, y_1, x_2, y_2 > 0$. Så gælder:

$$\begin{vmatrix} -x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 \\ x_2 & -y_2 \end{vmatrix} = \begin{vmatrix} -x_1 & -y_1 \\ -x_2 & y_2 \end{vmatrix} = \begin{vmatrix} x_1 & -y_1 \\ -x_2 & -y_2 \end{vmatrix} \Rightarrow \mathbf{D} < 0.$$

Og tilsvarende:

$$\begin{vmatrix} x_1 & y_1 \\ -x_2 & y_2 \end{vmatrix} = \begin{vmatrix} x_1 & -y_1 \\ x_2 & y_2 \end{vmatrix} = \begin{vmatrix} -x_1 & -y_1 \\ x_2 & -y_2 \end{vmatrix} = \begin{vmatrix} -x_1 & y_1 \\ -x_2 & -y_2 \end{vmatrix} \Rightarrow \mathbf{D} > 0.$$

Hvis et lige antal er negative, så er det enten de to diagonalelementer eller ikke. Hvis de to er diagonalelementer kan deres fortegn ændres uden at ændre $\mathbf{D}$s fortegn. Hvis ikke, er det to elementer i samme række eller søjle, og så ændres $\mathbf{D}$s fortegn hvis de to elementer ændrer fortegn. Hvis et element er 0, kan $\mathbf{D}$s fortegn udledes af fortegn for de to elementer i det andet produkt. Det gælder derudover at $x_2 \geq x_1$ og $y_2 \geq y_1$. Hvis $x_2 < x_1$ eller $y_2 < y_1$ kan man ombytte rækkerne, hvorved $\mathbf{D}$ skifter fortegn. I de resterende tilfælde mht. indbyrdes relation kan man let finde $\mathbf{D}$s fortegn:

$$x_2 \geq x_1 \text{ og } y_2 < y_1 \Rightarrow \mathbf{D} < 0,$$

og

$$x_1 > x_2 \text{ og } y_2 \geq y_1 \Rightarrow \mathbf{D} > 0.$$

Man kan altså antage følgende i det resterende:

$$x_1, y_1, x_2, y_2 > 0 \text{ og } x_2 \geq x_1 \text{ og } y_2 \geq y_1.$$

Herefter fås dette:

$$x_2 = x_1 k_1 + x_r, \text{ for } k_1 \in \mathbb{N}, 0 \leq x_r < x_1.$$

Man definerer derpå:

$$y_r = y_2 - y_1 k_1.$$

Det er altså en rækkeoperation man her beskriver, hvori række to fratrækkes k_1 gange række et:

$$\mathbf{D} = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 \\ x_2 - x_1 k_1 & y_2 - y_1 k_1 \end{vmatrix} = \begin{vmatrix} x_1 & y_1 \\ x_r & y_r \end{vmatrix}.$$

De enkelte elementer skal som sagt være mindre end 2^b, så der opstår overløb hvis $y_1 > 2^b/k_1$ (iflg. artiklen, men der burde såvidt jeg kan se stå $y_1 \geq 2^b/k_1$): $y_1 = 2^b/k_1 \Rightarrow y_2 - y_1 k_1 = y_2 - 2^b$. Det er dog ikke ødelæggende, da man i så fald ved at $\mathbf{D} < 0$ fordi:

$$y_r < 0, x_1 > 0, y_1 > 0 \text{ og } x_r \geq 0 \Rightarrow x_1 y_r < 0 \text{ og } y_1 x_r \geq 0.$$

Hvis $y_r > y_1$ kan fortegnet også nemt findes:

$$y_r > y_1 \Rightarrow \mathbf{D} > 0, \text{ da } x_r < x_1 \text{ og } y_1 < y_r.$$

I resterende tilfælde gør man en af følgende to ting:

$$x_r < \frac{x_1}{2} \text{ og } y_r < \frac{y_1}{2} \Rightarrow \mathbf{D} = \begin{vmatrix} x_1 & y_1 \\ x_r & y_r \end{vmatrix}.$$

$$x_r > \frac{x_1}{2} \text{ og } y_r > \frac{y_1}{2} \Rightarrow \mathbf{D} = \begin{vmatrix} x_1 & y_1 \\ x_1 - x_r & y_1 - y_r \end{vmatrix}.$$

I begge tilfælde benyttes kun sammenligninger og det der kaldes *euklidisk division* (dvs. heltalsdivision: $k_1 = \lfloor x_2/x_1 \rfloor$). Resultatet er en ny determinant hvor de to elementer i anden række som minimum er halveret. Dette vil enten blive gentaget eller algoritmen stopper. Dog garanteres det at den stopper efter maksimalt b iterationer.

Til at implementere orient2d-prædikatet, der tager tre punkter p, q, r som inddata, udregnes determinanten for denne 2×2-matrix:

$$\begin{bmatrix} p_x - r_x & p_y - r_y \\ q_x - r_x & q_y - r_y \end{bmatrix}.$$

3×3-matrix.
I dette tilfælde søger man fortegnet for denne determinant:

$$\mathbf{D} = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}.$$

Den grundlæggende ide er at man, medmindre fortegnet kan findes direkte, erstatter matricen med en anden der har samme determinant, men med bevisligt mindre elementer. Hertil bruges rækkeoperationer.

I matricen herover kaldes vektoren $(x_i, y_i, z_i)^T$ for U_i, men enhedsvektorerne langs akserne kaldes E_x, E_y, E_z. Lad u_i være projektionen af U_i på xy-planen. Det antages at u_1, u_2 er lineært uafhængige, hvilket medfører at U_3 kan udtrykkes som denne linearkombination:

$$U_3 = k_1 U_1 + k_2 U_2 + k_3 E_z \quad \text{for } k_1, k_2, k_3 \in \mathbb{R}.$$

Det medfører at:

$$\mathbf{D} = k_3 \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ 0 & 0 & 1 \end{vmatrix} = k_3 \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}.$$

Hvis fortegnet for k_3 er kendt skal man altså blot finde determinanten for en to-dimensionel matrix, hvilket kan gøres med metoden beskrevet herover. Artiklen viser herefter at k_3 i de fleste tilfælde kan bestemmes.

Til implementering af incircle-prædikatet, der tager fire punkter p, q, r, s som inddata, udregnes determinanten for denne matrix:

$$\begin{bmatrix} p_x - s_x & q_x - s_x & r_x - s_x \\ p_y - s_y & q_y - s_y & r_y - s_y \\ p_x^2 + p_y^2 - s_x^2 - s_y^2 & q_x^2 + q_y^2 - s_x^2 - s_y^2 & r_x^2 + r_y^2 - s_x^2 - s_y^2 \end{bmatrix}.$$

Denne metode er i øvrigt i en senere artikel [10] blevet generaliseret til $n \times n$ matricer.

4.3 Triangulering med CGAL

CGAL er et omfattende bibliotek til algoritmisk geometri med fokus på at udvikle effektive, pålidelige, robuste og fleksible algoritmer [11, 7].

Delaunay-triangulering i CGAL udføres med en inkrementel algoritme. Til punktlokalisering findes iflg. [7] to metoder, hop+vandring og Delaunay-hierarki, hvor den første er standardmetoden. I dokumentationen for CGAL [11] tilrådes det ved store punktmængder (> 10000 punkter iflg. [22, kap. 65]), at indkapsle Delaunay-klassen i et triangulerings-hierarki. Jeg anvender derfor Delaunay-hierarkiet.

4.3.1 ANN med CGAL

Som med TTL er det også relativt let at løse ANN med CGALs triangulering. Man danner blot en Delaunay-triangulering, og traverserer derefter strukturen knude for knude. For hver knude finder man den kvadrerede afstand til hver forbundet knude og gemmer den nærmeste. I CGAL anvendes formelt det der kaldes cirkulatorer, men i princippet gøres det samme som i TTL.

4.3.2 Kompleksitet

Som sagt benytter CGAL en inkrementel Delaunay-algoritme og i min implementering benyttes Delaunay-hierarki til punktlokalisering. Ifølge [14] konstrueres en Delaunaytriangulering for n punkter i planen i forventet $O(n \log n)$ tid. Som vist ovenfor tager ANN-søgningen $O(n)$ tid, så samlet er forventet tidskompleksitet $O(n \log n)$ for ANN med CGAL, men i værste fald også her $O(n^2)$.

4.3.3 Numerisk robusthed

CGAL består af tre dele: kerner, datastrukturer & algoritmer og diverse støttefunktioner. Kerner indeholder de primitive objekter som punkter, vektorer, linier, planer, trekanter m.m., samt operationer på disse. Valg af en kerne og en taltype (som er en template-parameter til kernen) indebærer valg af aritmetik og punktrepræsentation. Vælges fx den kerne der kaldes *Cartesian* med taltype

double, fås flydende tal-aritmetik med punkter fx i $\mathbb{R}^2$ repræsenteret ved to koordinater. Vælges Cartesian med taltypen *GMP*, GNU's multipræcisionstaltype [20] fås eksakte beregninger. Der findes også en kerne kaldet *Homogeneous*, hvori tal repræsenteres med homogene koordinater.

Valg af kerne styres til dels af hvad programmet laver, og en vigtig faktor er hvorvidt et program danner geometriske objekter som fx punkter, eller ikke. Delaunaytriangulering danner ingen nye objekter, og til det findes en prædefineret kerne kaldet *Exact Predicates, Inexact Constructions*. Denne kerne er blot et synonym for en filtreret kartesisk kerne med taltype double. Det er denne kerne jeg benytter til at lave en robust triangulering. Jeg har for sammenligningens skyld også lavet to versioner med ikke-filtreret kartesisk kerne: Den ene benytter taltypen double og er ikke robust, den anden benytter taltypen GMP [20] og er eksakt.

For at sikre robusthed benytter den filtrerede kerne semi-statiske filtre i nogle få prædikater, heriblandt orient2d og incircle, og dynamiske filtre til alle andre prædikater. Derudover benyttes i sidste ende eksakte taltyper hvis filtrene fejler. Det har ikke været muligt at finde en samlet eksakt beskrivelse af hvordan den filtrerede kerne er lavet, så det følgende er baseret på flere forskellige kilder, [36, 9, 11], og på gennemgang af kildekoden.

I CGALs inkrementelle Delaunay-triangulering med hierarki anvendes orient2d i punktlokalisering og incircle i lovliggørelse af kanter. Deraf konkluderer jeg at der mht. mit program er disse tre trin i filtret, hvor man benytter filter to hvis filter et fejler, og eksakt regning hvis filter to fejler (antagelsen støttes også af at [36] nævner disse tre trin):

1. Semi-statisk filter.
2. Dynamisk filter.
3. Eksakt beregning.

Semi-statisk filter

Det semi-statiske filter er beskrevet i [36] som danner grundlag for det følgende.

Man kan finde orienteringen for tre punkter p, q, r i planen ved hjælp af determinanten for en 3×3-matrix:

$$\begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = \begin{vmatrix} q_x - p_x & r_x - p_x \\ q_y - p_y & r_y - p_y \end{vmatrix} = pq_x \cdot pr_y - pq_y \cdot pr_x$$

Determinanten udregnes med flydende tals-operationer. I hvert af de to led i udtrykket findes den største absolutte faktor, $max_x = max(abs(pq_x), abs(pr_x))$ og $max_y = max(abs(pr_y), abs(pq_y))$. Man sikrer derpå at max_y indeholder den største af disse to. Den maksimale fejl udregnes, og størrelsen af max_x og evt. max_y undersøges. Herunder ses salgoritmen:

FILTRERET-ORIENT2D(p, q, r)

1 $det = pq_x \cdot pr_y - pq_y \cdot pr_x$
2 $max_x = max(abs(pq_x), abs(pr_x))$
3 $max_y = max(abs(pr_y), abs(pq_y))$
4 eps =(8.8872057372592798e-16)$\cdot max_x \cdot max_y$
5 **if** $(max_x > max_y)$ $ombyt(max_x, max_y)$
6
7 **if** (max_x <1e-146)
8 **if** $(max_x = 0)$ returner KOLLINEÆR
9 **elseif** (max_y <1e153)
10 **if** $(det > eps)$ returner POSITIV
11 **if** $(det < -eps)$ returner NEGATIV
12 Benyt mere præcis metode til udregning.

Hvis intet eksplicit resultat returneres ender algoritmen altså i l. 12 og derfra kaldes en mere præcis metode.

Hvis max_x er tilstrækkelig lille kan eps være i underløb, og man er så kun sikker på korrekt resultat hvis $max_x = 0$ (subnormale tal har en bit mindre præcision end normaliserede tal). Hvis $max_x = 0$ ligger de tre punkter på linie. Hvis ikke, benyttes en anden metode.
Hvis max_y ikke er stor nok til at det kan være i overløb undersøges det, som har det korrekte fortegn hvis den er tilstrækkeligt langt fra 0, dvs. hvis $|det| > eps$. I så fald returneres det relevante svar. Hvis $|det| \leq eps$ benyttes en anden metode. Hvis max_y er så stor at det kan være i overløb, benyttes en anden metode.
De tre konstanter i l. 4, 7 og 9 findes på følgende måde.

8.88...e-16:
Som nævnt er forskellen mellem et eksakt resultat x af en beregning med reelle tal og et resultat $\tilde{x}$ af en beregning med flydende tal begrænset sådan: $|x - \tilde{x}| \leq \varepsilon|x|$, hvor $\varepsilon = 2^{-53}$ i dobbelt præcision. I dette filter tages også hensyn til at visse processorer internt anvender udvidet præcision i visse situationer. Derfor anvendes $\varepsilon_{up} = 2^{-53} + 2^{-64}$ hvilket medfører at den maksimale fejl i determinanten er $eps = \varepsilon_{up} 2 max_x max_y = (2^{-52} + 2^{-63}) max_x max_y$. At konstanten rent faktisk er $\approx 2^{-50}$ kan jeg ikke forklare — at der skulle afrundes for at være på den sikre side virker ikke helt overbevisende. Der er i øvrigt en lille forskel mellem konstanten i artiklen og i koden. Den her viste er fra koden, mens der i artiklen står 8.8872057372592758e-16. Næstsidste ciffer er her 5, og 9 i koden.

1e-146:
I l. 7 undersøges max_x for at fange evt. underløb i $eps = \varepsilon_{up} max_x \cdot max_y$. Hvis $eps \geq 2^{-1022}$ undgås underløb:
$eps \geq 2^{-1022} = 2^{-50} 2^{-972} = \varepsilon 2^{-486^2} = \varepsilon max_x \cdot \max_y$ Derfor undersøges om $max_x < 10^{-146} \approx 2^{-486}$.

1e153:
I l. 9 undersøges max_y for at fange evt. overløb i determinanten. Lad $MAXDBL$ være det størst mulige tal i dobbelt præcision. Hvis det ikke må overløbe skal dette gælde:

$max_y < \sqrt{\frac{MAXDBL}{2}}$, fordi $det \leq 2 \times \max_y^2$. Derfor, hvis det flg. gælder er man sikret mod overløb i determinanten:
$max_y < \sqrt{\frac{MAXDBL}{2}} < \sqrt{\frac{2^{1024}}{2}} = \sqrt{2^{1023}} < 2^{512}$.
I programmet undersøges om $max_y < 10^{153} \approx 2^{508}$. Her er også en forskel mellem mit bud og artiklens som jeg ikke kan forklare — se ovenfor.

Et væsentligt element i [36] er at filtret verificeres (til dels automatisk) med et værktøj kaldet Gappa [19], en automatisk bevis-generator. I [36] vises at samme princip som ovenfor anvendes til incircle-prædikatet.

Dynamisk filter

Hvis det semi-statiske filter fejler vil et dynamisk filter blive brugt. Dette filter [9] benytter intervalregning og LU-faktorisering (dvs. Gauss-eliminering — se fx [27]). Der er her to metoder til udregning af determinantfortegn (og i øvrigt også determinantens værdi). Den naive (som den kaldes i [9]) beregner først $PA = [L][U]$ med brug af intervalregning, hvor A er den matrix hvis determinant man søger fortegnet for. Hvis et af intervallerne i diagonalen for $[U]$ indeholder 0 er svaret *VED IKKE*. Hvis ikke, returneres $det(P) \cdot \Pi_i fortegn([r_{i,i}])$ — her er $[r_{i,i}]$ et vilkårligt tal i det pågældende diagonalelement (som altså er et interval). Der skal ganges med $det(P)$ da en permuteret identitetsmatrix ikke nødvendigvis har determinant lig 1. Denne metode er bedst til lavdimensionelle matricer, så man kunne forestille sig at den kunne anvendes her, hvor man ved at punkter er i $\mathbb{R}^2$.

Der beskrives også i [9] en mere avanceret metode som kan anvendes på matricer af vilkårlig dimension. Den baseres på følgende argumentation.

$$PA = LU \Rightarrow A = P^{-1}LU \Rightarrow A^{-1} = (P^{-1}LU)^{-1} = U^{-1}L^{-1}P$$

Da man benytter flydende-tals-operationer fås kun tilnærmelser til LU, mens P kan udregnes eksakt. Derfor definerer man, baseret på L_{inv} og U_{inv} som numeriske tilnærmelser til L^{-1} og U^{-1}:

$$B := U_{inv}L_{inv}P$$

Hvis A er regulær, har L_{inv} stadig alle diagonal-elementer lig 1, og er stadig en nedre trekantmatrix, derfor er $det(L_{inv}) = 1$. U_{inv} har i diagonalen de inverse af U's diagonalelementer. Da følgende gælder:

$$det(A) = det(B^{-1}BA) = det(B^{-1})det(BA)$$

får man at

$$det(BA) > 0 \Rightarrow fortegn(det(A)) = fortegn(det(B^{-1})) = fortegn(det(B))$$

Det sidste lighedstegn gælder fordi:

$$det(B^{-1})det(B) = det(B^{-1}B) = det(I) = 1$$

Man skal altså bevise at $det(BA) > 0$, og hvis det er umuligt svares VED IKKE. Metoden benytter dette resultat (hvor matrix-normen kan vælges vilkårligt):

$||F|| < 1 \Rightarrow ||(I+F)^{-}1|| \leq \frac{1}{1-||F||}$ og $det(I+F) > 0$.

Her beskrives blot de efterfølgende trin. Matrix-normen $||\cdot||_\infty$ er den største absolutte rækkesum for matricen.

1. Udregn $PA \approx LU$ med flydende-tal-operationer. Hvis dette er umuligt (oftest pga. en singulær matrix) svares VED IKKE.
2. Udregn U_{inv} og L_{inv}, som er numeriske tilnærmelser til U og L. Hvis der er overløb i en eksponent svares VED IKKE.
3. Udregn $|||[I - BA]|||_\infty$ med intervalregning.
4. Hvis $|||[I-BA]|||_\infty < 1$ returneres $det(P)\Pi_i fortegn(u_{ii})$. Ellers svares VED IKKE.

Eksakt beregning

Hvis det dynamiske filter svarer VED IKKE udregnes resultatet med en eksakt metode. I den filtrerede kerne, som håndterer dette og derfor skal have en udvej hvis man falder tilbage til eksakte beregninger, benyttes det der kaldes *Exact_type_selecter* til valg af den eksakte type. Standardvalget er MP_float, som iflg. [11] kan repræsentere et flydende tal med vilkårlig præcision, bortset fra at eksponenten kan løbe over, da den er repræsenteret ved en double.

5

Plane sweep

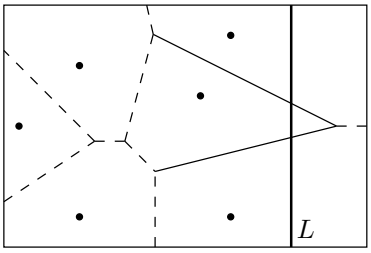
I [28] beskrives en *plane sweep*-algoritme, som den følgende gennemgang er baseret på. Inddata til denne algoritme forventes at være heltalligt.

5.1 Algoritmen

Algoritmen består overordnet af tre faser:

1. En lodret linie L fejer over planen fra venstre mod højre, og behandler undervejs hvert punkt (hændelse). Behandlingen af et punkt p består i at finde dets nærmeste nabo til venstre, $v = \mathrm{NN}_v(p)$. Det samlede resultat af denne fase kaldes $\mathrm{NN}_v(S)$.
2. Som i fase et, blot fejes her fra højre mod venstre, og man finder hvert punkts nærmeste nabo til højre, $h = \mathrm{NN}_h(p)$. Det samlede resultat af denne fase kaldes $\mathrm{NN}_h(S)$.
3. Til sidst sammenlignes $\mathrm{NN}_v(p)$ med $\mathrm{NN}_h(p)$, for alle $p \in S$. Det nærmeste vælges.

Fase et og to er symmetriske, og for at gøre programmet enklere, udfører jeg fase to ved at spejle punkterne i y-aksen, finde NN_v for de spejlvendte punkter og derpå spejle resultatet tilbage. Af disse grunde beskrives her kun fase et.



Figur 5.1: *Et Voronoi-diagram for seks punkter. To bisektorer klassedeler L i tre intervaller.*

Tegnes en lodret linie, L, ned gennem et Voronoi-diagram for S, $\mathrm{Vor}(S)$, ses at L inddeles i intervaller af Voronoi-kanterne, se fig. 5.1. Til hvert interval hører

netop én Voronoi-region, og altså netop ét punkt i S. Et vilkårligt punkt p på L er i netop ét sådant interval, og man har dermed fastlagt den nærmeste nabo for p, nemlig generatoren for regionen. Dette udnyttes her, med den modifikation at man behandler S to gange, idet man danner $\mathrm{NN}_v(S)$ og $\mathrm{NN}_h(S)$. Fordelen er at man kun skal vedligeholde et delvist Voronoi-diagram, nemlig diagrammet for den delmængde af de hidtil sete punkter, hvis kanter krydser L, i fig. 5.1 de fuldt optrukne linier. Kanterne er (stykker af) *bisektorer* for punkterne i S. En bisektor for to punkter p og q, *bs(p,q)*, er den linie der, vinkelret på liniestykket pq, løber midt mellem p og q. Punktet med det mindste y-koordinat definerer jeg som bisektorens nedre punkt, og det andet det øvre. For lodrette bisektorer er punktet med størst x-koordinat det nedre. En bisektor for to punkter med identiske koordinater vil blive opfattet som en lodret bisektor, og vil stadig have et nedre og et øvre punkt. En bisektor er fuldt beskrevet ved dens to punkter, dvs. har man to punkter, har man en bisektor og omvendt.

De to primære datastrukturer er en prioritetskø og et binært søgetræ. Indledningsvist indsættes alle punkter i S, også kaldet datapunkter, i prioritetskøen Q. Herefter fejes henover S idet man udtager punkterne et for et. Det binære søgetræ, T, indeholder bisektorer, og det bruges til at finde $\mathrm{NN}_v(p)$. Bisektorerne er i træet ordnet efter deres y-værdi på L (mere om ordningen senere). Når et punkt p udtages fra Q, findes, ved søgning i T, de to bisektorer der omgiver p på L: Den øvre og den nedre *grænse*[1] for p, i form af bisektorer. Man har nu $\mathrm{NN}_v(p)$ ved enten den øvre grænses nedre punkt eller den nedre grænses øvre punkt. Når dette er gjort, skal T opdateres for at afspejle det nye punkts tilstedeværelse. Dette medfører at der indsættes en eller to nye bisektorer i T.

Når to bisektorer skærer hinanden, inverteres deres indbyrdes ordning i skæringspunktet. Dette håndteres ved at undersøge for skæringspunkter hver gang en bisektor er blevet indsat i træet. Hvis den nye bisektor skærer en eller begge naboer (i T), indsættes skæringspunktet i prioritetskøen, og bliver altså en hændelse. Sammen med skæringspunktet gemmes også de to bisektorer, kaldet *forældrene*, der forårsager skæringen. Pseudokode for fase et ser sådan ud:

PLANE-SWEEP-FASE-ET(S)

```
1  while (Q not empty)
2     p ← Q.extractMin()
3     if (p er et datapunkt)
4           (v, øvreL, nedreL) ← T.findNN_v(p)
5           opdaterTræ(p, v, øvreL)
6           opdaterTræ(p, v, nedreL)
7     elseif (p er skæringspunkt og forældre-bisektorer ∈ T)
8           Dan erstatningsbisektor ERB
9           T.slet(forælder-A)
10          T.slet(forælder-B)
11          T.indsæt(ERB)
12          checkForSkæring(ERB)
```

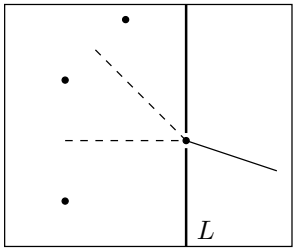
[1] Grænser er altså her bisektorer, og er naturligvis grænser for det interval p er i, og ikke for punktet, men det letter forklaringen at misbruge terminologien på den måde.

T antages at være globalt tilgængelig. Funktionen findNN$_v$ returnerer tre ting: NN$_v(p)$, en iterator til bisektorer over p, og en iterator til bisektorer under p; de to sidstnævnte kaldet hhv. øvreL og nedreL herover. Iteratorerne giver adgang til en liste af bisektorer, hvor bisektoren umiddelbart over hhv. under p er første element i øvreL hhv. nedreL. Man har dermed også adgang til en liste af punkter. Dette bruges i forbindelse med opdatering af T. Punktet p kan ligge så yderligt at der enten ingen øvre eller ingen nedre bisektor er, men en af dem vil altid findes[2]. Funktionen checkForSkæring gør som beskrevet over pseudokoden, og skal naturligvis også kaldes når en erstatning for to krydsende bisektorer indsættes.

Et punkt der udtages fra Q er altså enten et skæringspunkt eller et datapunkt. Her forklares hvordan man behandler hvert tilfælde.

5.1.1 Behandling af et skæringspunkt

Hvis to bisektorer der skærer hinanden, stadig begge findes i søgetræet når skæringspunktet nås, slettes de og en ny indsættes. Denne nye er bisektor mellem de to gamles ekstreme punkter. I fig. 5.2 ses at de to stiplede bisektorer skærer hinanden. Da den sidste af disse bisektorer blev indsat i T, blev skæringspunktet indsat i køen. Ved behandling af skæringspunktet erstattes de to stiplede bisektorer af bisektoren mellem den øvre bisektors øvre punkt og den nedre bisektors nedre punkt.



Figur 5.2: *Skæringspunktet bevirker at to bisektorer erstattes med en ny.*

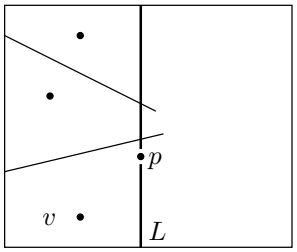
5.1.2 Behandling af et datapunkt

Når et datapunkt p behandles, findes som sagt den øvre og/eller nedre grænse for punktets interval, og man har nu NN$_v(p) = v$, se fig. 5.3. Herefter skal søgetræet opdateres, så det afspejler p's tilstedeværelse.

Opdateringen går ud på at finde den eller de to kanter (bisektorer) i p's Voronoi-region der krydser L. Det betyder at et eller flere andre intervaller ændres. Hvis p ligger yderligt behøves kun en ny bisektor i T, ellers kræves to. Opdatering foregår i to trin, et opad og et nedad. De to trin er symmetriske, så kun opdatering opad beskrives i detaljer.

Udgangspunktet for opdatering opad er dette: Man har p, v og listen af bisektorer over v. Den bisektor der til enhver tid peges på i denne liste, kaldes for

[2]Dette kan jeg garantere, idet jeg eksplicit behandler de to første punkter, og dermed indsætter den første bisektor.

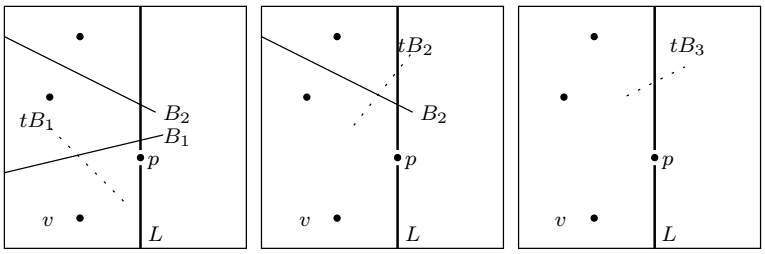


Figur 5.3: *Et nyt punkt p behandles. Dets NN_v er v. øvreL peger på bisektorerne over p. p har her ingen nedre bisektor.*

den *aktuelle*, og det er altså den øvre grænse når opdateringen starter. Tilsvarende er det *aktuelle* punkt den aktuelle bisektors nedre punkt. Opdateringen har som mål at finde den bisektor i p's Voronoi-region der krydser L over p, hvis der findes en sådan. Det sker ved at danne temporære bisektorer, tB, mellem p og de eksisterende punkter, startende med $bs(p, v)$, og evt. slette eksisterende bisektorer. tB krydser enten L i intervallet mellem p og den aktuelle bisektor, eller uden for intervallet (hvis tB er lodret er den også uden for intervallet).

I fig. 5.4 ses hvordan opdateringen forløber: Den første temporære bisektor, tB_1, skærer ikke L mellem p og B_1. Derfor slettes B_1 og tB_2 dannes. tB_2 krydser ikke L mellem p og B_2, så B_2 slettes og tB_3 dannes. Der er nu ikke flere eksisterende bisektorer, og løkken stopper. Til sidst i opdateringen undersøges om den senest dannede temporære bisektor krydser L over p, hvilket er tilfældet her. Derfor indsættes tB_3 i T.

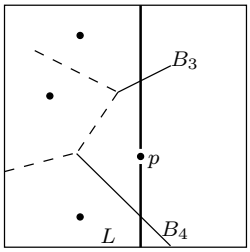
Opdatering nedad: Der er ingen eksisterende bisektorer under p, så opdateringen består kun i at danne den første temporære bisektor, $bs(p, v)$, og kontrollere om den krydser L under p. Da det er tilfældet, indsættes den i T.



Figur 5.4: *Undervejs i opdatering opad dannes et antal temporære bisektorer, her tB_1, tB_2 og tB_3. Der indsættes nul eller en bisektor under opdatering opad.*

I fig. 5.5 ses resultatet af fuld opdatering efter p. Kun de to fuldt optrukne kanter er med i T. Det ses at ikke alle kanter fra det rigtige Voronoi-diagram gemmes i T. Den bisektor der forbinder B_3 og B_4 eksisterede kun midlertidigt, i opdateringsfasen, men blev forkastet da den krydsede L over B_2.

Pseudokode for opdatering opad ser sådan ud:



Figur 5.5: *Slutresultat efter behandling af det nye punkt p. To eksisterende bisektorer blev slettet, og to nye indsat.*

OpdaterTræ(p, q, bL)

```
1  tB ← bs(p, q)
2  eB ← bL->Bisektor
3  while (bL peger på valid bisektor og tB.y ∉ interval(p.y, eB.y))
4      q ← eB.øvrepunkt
5      tB ← bs(p, q)
6      bL++
7      T.slet(eB)
8      eB ← bL->Bisektor
9  if (flere aktive punkter eller tB.y > p.y)
10     T.indsæt(tB)
11     checkForSkæring(tB)
```

Da bL er en iterator til listen af de øvre bisektorer, kan den inkrementeres (bL++), hvilket giver den næste i træet. Udtrykket bL->Bisektor returnerer den bisektor iteratoren peger på. Når enden af listen nås, peger bL ikke mere på en valid bisektor. Udtrykket p.y står for p's y-koordinat, lige som eB.y og tB.y står for den eksisterende hhv. den temporære bisektors y-værdi på L.

5.2 Kompleksitet

Samlet tidskompleksitet for algoritmen er $O(n \log n)$.

Skæringspunkter: Der indsættes altid en eller to nye bisektorer i søgetræet T pr. datapunkt: En efter opdatering opad og/eller en efter opdatering nedad. Der er kun grund til at undersøge for skæring med en ny bisektors to naboer i T; Voronoi-diagrammet kan ses som en planar graf [6, s. 150]. Hvis en bisektor kunne krydse både sin egen øvre nabo og dennes øvre nabo, ville man ikke længere have en planar graf. Jeg undersøger om skæringspunkter opstår på denne måde:

- Hvis kun en ny bisektor blev indsat efter opdatering, undersøges for skæring mellem den nye bisektor og dens øvre og dens nedre nabo i træet.
- Hvis to nye bisektorer blev indsat efter opdatering, B_o efter opdatering opad og B_n efter opdatering nedad, undersøges for skæringspunkt mellem B_o og dennes øvre nabo, mellem B_n og B_o, og mellem B_n og dennes nedre nabo.

Der kan altså højest indsættes tre skæringspunkter for hvert nyt datapunkt, ialt $3n$.

Hver gang et skæringspunkt behandles, indsættes en ny bisektor. Dette medfører at der kan opstå op til to nye skæringspunkter. Behandling af et skæringspunkt medfører at et datapunkt deaktiveres, hvilket vil sige at det højest kan forekomme n gange, så her kan der altså indsættes $2n$ skæringspunkter, totalt giver det $5n$ indsættelser af skæringspunkter. Den indledende opbygning af Q tager $O(n)$, mens udtagning af mindste element tager $O(\log n)$; i alt giver det en tidskompleksitet mht. Q på $O(n \log n)$.

Bisektorer: I alt indsættes maksimalt $O(n)$ bisektorer, da et Voronoi-diagram som nævnt er en planar graf, og antal kanter (dvs. bisektorer) derfor er lig $O(n)$ se [6, teorem 7.3, s. 150]. Der kan derfor heller ikke slettes flere end $O(n)$ bisektorer. Jeg laver en søgning pr. datapunkt og to pr. skæringspunkt. Disse operationer tager alle $O(\log n)$, så i alt er tidskompleksit også her $O(n \log n)$.

Det ovenfor beskrevne er for gennemførelse af fase et, så det skal ganges med to. Dertil kommer fase tre, som tager $O(n)$ (for hvert punkt i S sammenlignes to tal), så den samlede kompleksitet er som nævnt $O(n \log n)$.

5.3 Algoritmisk robusthed

5.3.1 Skæringspunkter

Skæringspunkter der falder uden for rektanglet der udgøres af fejelinien, det største x-, og det mindste og største y-koordinat, kan ignoreres. Ses fig. 5.7 som delvise Voronoi-diagrammer, er det klart at hele rektanglet som defineret herover, for begge figurer ligger i Voronoi-regionen tilhørende B_2's nedre hhv. øvre punkt, for figuren til venstre hhv. til højre. Der er derfor ingen grund til at bruge tid på skæringspunkter uden for rektanglet. Dette skal indbygges i bisektorernes ordning, som det ses i tilfælde 1(b) og 1(c) herunder.

5.3.2 Bisektorernes ordning

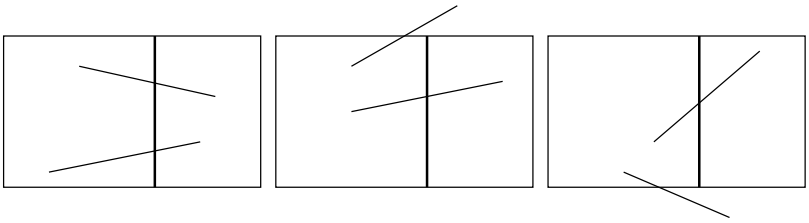
For at sammenligne to bisektorer, B_1 og B_2, kræves et x-koordinat, men jeg kan med brug af C++-standardbiblioteket ikke angive dette x i sammenligninger. Jeg løser det problem ved, i enhver sammenligning at benytte det største x-koordinat fra de fire punkter der udgør de to bisektorer (en af grundene til kun at feje i en retning). For en punktmængde S, en bisektor B_n og et x-koordinat har man denne information:

y_n	:	y-værdi for B_n i et x fundet som beskrevet herover.
h_n	:	hældning for B_n.
$minX_n, maxX_n$	:	mindste hhv. største x-koordinat for B_n.
a_n, b_n, c_n	:	koefficienterne for B_n.
$minY, minY$	:	mindste hhv. største y-koordinat for S.

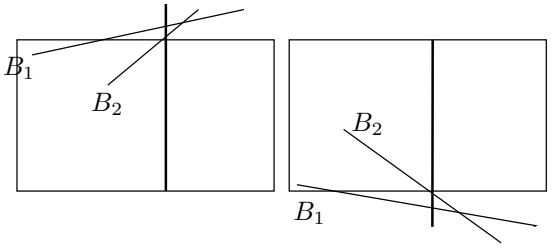
Ordningen mellem to bisektorer, $B_1 < B_2$, kan nu afgøres på følgende måde (at B_1 er mindre end B_2 betyder at B_1 ligger under B_2 for det relevante x-koordinat):

1. Hvis hverken B_1 eller B_2 er lodret, og $y_1 \neq y_2$:
   (a) $y_1 \in [minY, maxY]$ eller $y_2 \in [minY, maxY] \Rightarrow$
   Bisektor med mindst y-værdi er mindst.
   (b) $y_1 > maxY$ og $y_2 > maxY \Rightarrow$
   Bisektor med størst $maxX$ er mindst.
   (c) $y_1 < minY$ og $y_2 < minY \Rightarrow$
   Bisektor med mindst $maxX$ er mindst.
2. Hvis hverken B_1 eller B_2 er lodret, og $y_1 = y_2$:
   (a) $h_1 \neq h_2 \Rightarrow$ Bisektor med mindst hældning er størst.
   (b) $h_1 = h_2$ og $maxX_1 \neq maxX_2 \Rightarrow$
   Bisektor med mindst $maxX$ er mindst.
   (c) $h_1 = h_2$ og $maxX_1 = maxX_2 \Rightarrow$
   Bisektor med størst $minX$ er mindst.
3. Hvis en bisektor er lodret og den anden ikke, er den lodrette den øverste.
4. Hvis begge er lodrette, så er den bisektor der ligger længst til venstre den øverste.

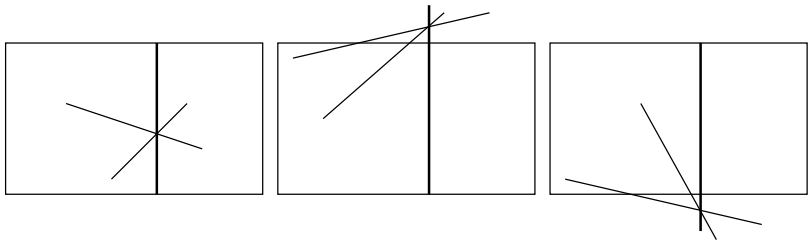
Herunder er tilfældene i pkt. 1 og 2 illustreret. Rektanglerne er her de mindste rektangler der kan dannes uden om punktmængden S, mens den lodrette linie er fejelinien.



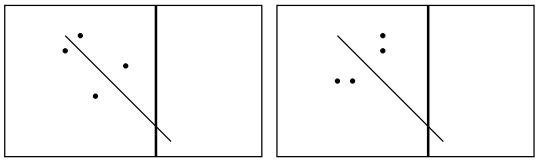
Figur 5.6: *Tilfælde 1 (a).*



Figur 5.7: *Til venstre tilfælde 1 (b), til højre 1 (c).*



Figur 5.8: *Tilfælde 2 (a).*



Figur 5.9: *Til venstre tilfælde 2 (b), til højre 2 (c).*

Kommentar til artiklen

I artiklen [28, s. 450] findes pseudokode for den funktion der kaldes *updateY*. Den opdaterer T, som beskrevet ovenfor. Sidste del består af en *if*-sætning der undersøger om den senest dannede temporære bisektor skal indsættes i T. I pseudokoden sker dette, hvis det besluttes ikke at indsætte en ny bisektor:

$$\texttt{Ytable}_{direction}\ \texttt{:= p}$$

Det står flere steder i artiklen at det der kaldes `Ytable` er et binært søgetræ der indeholder formler for lige linier, så jeg har svært ved at forstå denne sætning. Det giver ikke mening at indsætte et punkt i træet. Jeg har heller ikke kunnet finde en forklaring i teksten.

5.4 Numerisk robusthed

Artiklen [28] giver to bud på hvordan man kan sikre sig mod numeriske problemer. Det ene går ud på at regne med flydende tal, og fange problemer med overløb via *interrupt*. Det andet går ud på at regne eksakt. Overordnet set er det også de muligheder der angives i en artikel som [40, s.601].

Det første forslag er til dels motiveret af at algoritmen opfører sig meget pænt, ikke mindst fordi inddata skal være heltal. Der er ingen gentagne beregninger på samme variabel, så der forekommer ingen akkumulering af evt. fejl. Det eneste der konstrueres er skæringspunkter, og disse bruges kun til at gøre opmærksom på at to bisektorer skal erstattes med en tredie. Skæringspunktets koordinater indgår ikke i dannelsen af erstatningsbisektoren. Koefficienter for bisektorer kan udregnes uden afrundingsfejl, men kan forårsage overløb. Udregning af skæringspunkt mellem bisektorer og fejelinien (dvs. y-værdi for en

bisektor i et givet x) kan give afrundingsfejl og overløb.

Ved at ignorere skæringspunkter der ligger uden for rektanglet defineret af fejelinien, det største x-koordinat og det mindste og største y-koordinat i punktmængden, undgår man at skulle håndtere et antal situationer, og sparer derved tid og plads (i køen). Man undgår også at skulle opbevare skæringspunkter for nær-parallelle bisektorer, som vil kunne blive store.

Hvis man indbygger rutiner der fanger overløb kan en version af denne algoritme, med brug af *double*-typer og standard flydende-tals-beregninger, i mange tilfælde levere et korrekt resultat.

Det er dog muligt at få et fejlagtigt resultat. Ved eksperimenter er jeg stødt på en punktmængde hvor følgende sker. Ved behandling af et tidligere indsat skæringspunkt, $p = (p_x, p_y)$, indsættes en bisektor som erstatning for to andre. Erstatningen undersøges for skæring med naboer, og der findes et skæringspunkt, $q = (q_x, q_y, q_d) = (\frac{q_x}{q_d}, \frac{q_y}{q_d})$ (skæringspunkter udregnes som homogene koordinater). Som omtalt undersøger programmet om q kan ignoreres. I første omgang benyttede jeg q_d til multiplikation for at undgå division, i den tro at det øgede sandsynligheden for korrekte resultater. Det vil sige at q skal opfylde disse krav:

$$p_x q_d <= q_x <= maxX q_d$$

$$minY q_d <= q_y <= maxY q_d$$

Det viste sig i dette tilfælde at $q_x = p_x$, dvs. skæringspunktet ligger på fejelinien, og q skal derfor indsættes i køen og behandles. Problemet opstod da den første test svarede at $q_x < p_x q_d$, så skæringen ignoreres, og derved korrumperes søgetræet. Løsningen var at udføre sammenligningerne med q som et 2-koordinats-punkt og ikke med homogene koordinater. Altså:

$$p_x <= q_x/q_d <= maxX$$

$$minY <= q_y/q_d <= maxY$$

Man kan hævde at det er et pseudo-problem, idet jeg burde have vidst at det ville være mere korrekt at sammenligne flydende tal med flydende tal, og ikke flydende tal med rationelle tal. Det flydende tal kan dog i sig selv være en tilnærmet størrelse, så metoden er altså stadig ikke garanteret at returnere et korrekt resultat. En sådan garanti kan til gengæld opnås med den følgende metode.

Artiklen benytter rationelle tal til at opnå eksakte beregninger. En sådan type består af en tæller og en nævner. I LEDA [34] findes typen *leda::rational*, hvor både nævner og tæller er af typen *leda::integer*. Denne heltals-type er ubegrænset i størrelse, og giver ikke overløb, dvs. man har mulighed for i princippet at få ubegrænset præcision. Prisen er at hastigheden falder betragteligt. Iflg. LEDAs website [2] er *leda::integer* og *leda::rational* 30-100 gange langsommere end double.

Kan man angive den maksimale præcision, og findes en type der kan indeholde denne, så er dette at foretrække frem for ubegrænsede typer som ovennævnte. I [28] angives den maksimale præcision at være $9 + 5m$ bits, hvor m er det antal

bits der kræves for at repræsentere inddata. Dette kræver analyse af alle beregninger i programmet, og artiklen [28] nævner fire typer — disse findes også i min implementering, og derudover udregner jeg i sidste fase den kvadrerede afstand mellem to punkter. Det gælder for inddata (datapunkter), (x, y), at:

$$x, y, M \in \mathbb{Z}, |x| < M, |y| < M.$$

De beregninger jeg foretager er:

1. Beregning af koefficienter for linieligningen, $ax + by = c$. En bisektor dannes af to punkter (x_1, y_1) og (x_2, y_2), og koefficienterne fås sådan:
   $a = 2(x_1 - x_2), b = 2(y_1 - y_2) \Rightarrow a < 4M, b < 4M$
   $c = x_1^2 - x_2^2 + y_1^2 - y_2^2 \Rightarrow c < 2M^2$

2. Beregning af skæringspunkt for to bisektorer dannet af disse to punktpar $(x_1, y_1), (x_2, y_2)$ og $(x_2, y_2), (x_3, y_3)$. Skæringspunktet består af 3 tal, (s_x, s_y, s_d), dvs. i homogene koordinater, og udregnes som følger:
   $s_x = (x_1^2 + y_1^2)(y_2 - y_3) + (x_2^2 + y_2^2)(y_3 - y_1) + (x_3^2 + y_3^2)(y_1 - y_2) \Rightarrow$
   $s_x < 3(M^2 + M^2)(M + M) = 3(2M^2 2M) = 12M^3$
   s_y udregnes på samme måde og har samme øvre maksimum.
   $s_d = 2(x_1 y_2 - x_2 y_1 + x_2 y_3 - x_3 y_2 + x_3 y_1 - x_1 y_3) \Rightarrow$
   $s_d < 2(M^2 + M^2 + M^2 + M^2 + M^2 + M^2) = 12M^2$

3. Sammenligning af to bisektorer i et punkt x. Artiklen angiver en sammenligningsmetode der undgår division. Der indgår to bisektorer: $a_1 x + b_1 y = c_1$ og $a_2 x + b_2 y = c_2$. Disse to sammenlignes i punktet $x_0 = \frac{p}{q}, q \neq 0$. Sammenligningen sker ved denne udregning:
   $b_2(qc_1 - pa_1) < b_1(qc_2 - pa_2)$.
   Hvis sammenligningen sker i et skæringspunkt gælder at $|p| < 12M^3, |q| < 12M^2$. Derfor fås følgende resultat:
   $b_2(qc_1 - pa_1) < 4M(12M^2 \times 2M^2 + 12M^3 \times 4M) = 4M \times 72M^4 = 288M^5$.
   Hvis sammenligning sker i et datapunkt, gælder:
   $b_2(qc_1 - pa_1) < 4M(2M^2 + M \times 4M) = 24M^3$.

4. Sammenligning af punkter i prioritetskøen sker ved krydsmultiplikation af de indgående rationelle tal: $\frac{a}{b} < \frac{c}{d} m \Leftrightarrow ad < cb$. Skæringspunkter kan forårsage det største tal her, idet nævner og tæller er begrænset af hhv $12M^3$ og $12M^2$. Øvre grænse er altså $12M^3 12M^2 = 144M^5$.

5. Kvadreret afstand mellem to punkter:
   $kdist(p_1, p_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2 \Rightarrow |kdist(p_1, p_2)| < 4M^2$.

Samlet er altså $288M^5$ en øvre grænse for hvor store tal kan blive i denne algoritme, hvis man anvender artiklens måde at sammenligne bisektorer på. Hvis M kan repræsenteres med m bits, får man at $\lceil \log_2(288) \rceil + 5m$ bits er tilstrækkelig til at algoritmen giver eksakte resultater.

I min implementering er der fire situationer der kan volde problemer:

1. Beregning af y-værdi for en linie: $ax + by = c \Rightarrow y = (-ax + c)/b$.

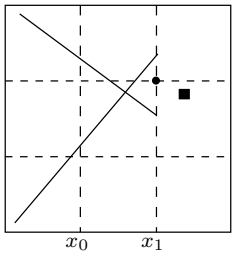
2. Beregning af skæringspunkt mellem to bisektorer.

3. Afgørelse af om et skæringspunkt er i rektanglet dannet af fejelinien, største $x \in S$, mindste $y \in S$ og største $y \in S$.

4. Overløb ved beregning af afstand mellem to punkter.

Alle fire punkter kan løses med brug af rationelle tal og artiklens metode til at sammenligne y-værdier. Hvis man kan garantere at både nævner og tæller er tilstrækkeligt små, kan man fx benytte Boost's rationelle type [8], hvor nævner og tæller er parametriseret. Dette medfører at man kan benytte fx den indbyggede *int* som nævner og tæller, hvilket alt andet lige vil være hurtigere end løsningen med *leda::rational*.

Ved implementering med flydende tal som koordinattype, og ved brug af rationelle tal til skæringspunkter, vil man kunne fjerne problemerne med afrunding på følgende måde:
1) Skæring med y-aksen kan udregnes som homogene kordinater.
2) Et skæringspunkt skal, for at give nøjagtighedsproblemer, fejlagtigt udregnes til at være på 'den anden side' af et heltal i forhold til det eksakte skæringspunkt. I fig. 5.10 ville det betyde at det approksimerede skæringspunkt skulle have x-koordinat større end x_1, fx hvor firkanten ligger, hvor det korrekte har x-koordinat mindre end x_1. I realiteten kan ethvert skæringspunkt rundes op[3], så det lander på næste hele tal, da ordningen i prioritetskøen gør at skæringspunkter behandles før datapunkter. Da inddata er heltalligt findes ingen datapunkter mellem to heltals-gridlinier, og pga. ordningen i prioritets-køen kan man runde op, og derved opnå kun at gemme heltal i prioritetskøen. Dette vil fjerne nogle af problemerne. Der vil ikke opstå et problem som antydet i fig. 5.10.



Figur 5.10: *Hvor galt kan det gå når et skæringspunkt udregnes?* $x_0, x_1 \in \mathbb{Z}$ *og* $x_1 = x_0 + 1$.

Et skæringspunkt, (s_x, s_y), udregnet i flydende-tal, kan ikke være så forkert, at det faktiske resultat har $s_x > x_1$ mens det faktiske skæringspunkt (i en ideel verden med reelle tal) har $x_0 < s_x < x_1$. Godt nok kan fejlen være op til $\frac{1}{2}ulp$, hvilket vil sige 0.5 hvis koordinater er så store heltal at alle bits er i brug, men hvis der anvendes *eksakt afrunding* vil resultatet af en operation på flydende tal altid give den nærmeste flydende talværdi der kan repræsenteres, og derfor kan en udregning ikke 'springe over' et heltal.
3) Da skæringspunkter er i homogene koordinater er der ingen afrundingsproblemer her.

[3] I [40] nævnes begrebet *snap rounding*, der såvidt jeg kan se går ud på at liniers endepunkter tvinges til at ligge på et *grid*.

4) Overløb er det eneste tilbageværende problem. Det må håndteres ved *interrupts* og et evt. skift til ubegrænsede typer, dvs. en filter-metode.

6

Kd-træer

Et Kd-træ [4] er et binært søgetræ til organisering af K-dimensionelle data, med henblik på forskellige former for søgning. I denne sammenhæng ses kun på to dimensioner hvor data er punkter i planen, men Kd-træer kan organisere vilkårligt mange dimensioner. Iflg. [4] er et Kd-træ velegnet til fx *range-*[1] og allenærmeste-nabo-søgninger. I [4] nævnes ANN-søgninger, men selve teksten er af pladshensyn slettet fra artiklen og der henvises i stedet til [18]. I [5] findes dog en i mine øjne klarere og kortere beskrivelse af algoritmer til både konstruktion af Kd-træer og ANN-søgning i dem.

6.1 Algoritmen

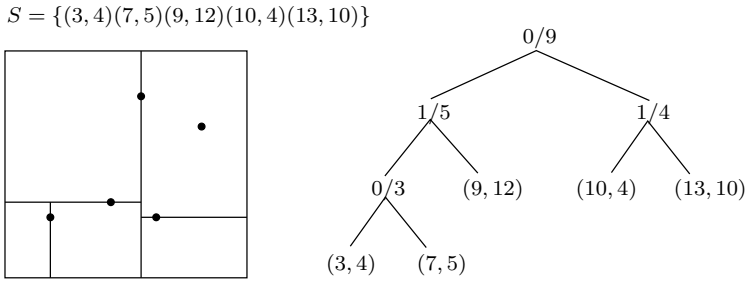
Kd-træer anvendt til ANN-søgning består strengt taget af to algoritmer, nemlig en der konstruerer træet og en der foretager søgningerne i træet.

6.1.1 Konstruktion

Lad punktmængden S, bestående af n K-dimensionelle punkter, være de data der organiseres. Et Kd-træ består af indre knuder og blade. Hver indre knude repræsenterer en delmængde af S mens hvert blad indeholder et punkt. Roden opdeler S i to omtrentligt lige store delmængder, kaldet venstre hhv. højre barn. Disse opdeles hver især igen i to halvdele og så fremdeles, se fig. 6.1.

Hver indre knude indeholder udover to børn også to tal, en diskriminator d og en partitionsværdi v. Diskriminatoren d angiver hvilken dimension den indre knude opdeler delmængden i, og partitionsværdien er det element i dimension d der mest ligeligt opdeler delmængden i to. Diskriminatoren kan vælges på forskellige måder, fx kan man cykle gennem de K dimensioner i rækkefølge. Dette er metoden i [6] hvor Kd-træer dog kun anvendes til range-søgninger. I [5] findes d som den dimension hvor spredningen af værdierne er størst, dvs. hvor forskellen mellem maksimum- og minimum-værdi er størst. Det er denne metode jeg benytter. Som partitionsværdi benyttes medianen for den aktuelle delmængde i dimension d. Se fig. 6.1 for et eksempel på en punktmængde og

[1] Fx finder en range-søgning i planen alle punkter der måtte befinde sig i et rektangulært område i planen angivet med to punkter, øverste højre og nederste venstre hjørne — se [6].



Figur 6.1: *Til venstre opdelingen af planen, til højre det tilhørende Kd-træ hvor indre knuder angives med d/v mens et blad viser punktet det indeholder, (x, y).*

dens Kd-træ, dannet som beskrevet. Pseudokode til konstruktion af et Kd-træ følger her.

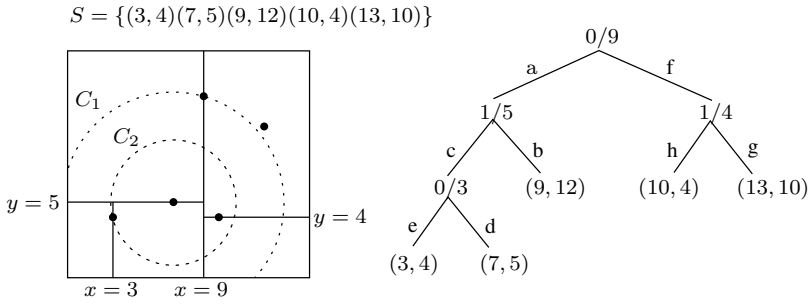
```
DAN-KDTRÆ(S, ni, øi)
1  k ← ny KD-knude
2  if (ni = øi)
3      k.bladdata ← S[ni]
4  else
5      k.d ← dimension med størst spredning i S[ni...øi]
6      k.v ← median for S[ni...øi] i dimension d
7      k.venstrebarn ← Dan-KDTree(S,ni, k.m)
8      k.højrebarn ← Dan-KDTree(S,k.m+1,øi)
9  Returner k
```

Konstruktionen startes med kaldet DAN-KDTRÆ$(S, 0, n-1)$ hvor S er den punktmængde man organiserer. De to tal, 0 og $n-1$, er hhv. nedre og øvre index for den aktuelle delmængde, altså hele S i første kald.

6.1.2 ANN-søgning

Søgning efter nærmeste nabo NN(p) for et punkt p gøres med en rekursiv funktion, som først kaldes med træets rod som argument. Funktionen vælger først det barn der indeholder p. Når funktionen når til et blad udregnes den kvadrerede afstand fra p til bladets punkt. Er dette punkt nærmere p end det hidtil nærmeste punkt, gemmes det nye punkt og den nye afstand. Efter behandling af det først valgte barn returneres til dettes forælderknude, hvor det skal afgøres om også det andet barn skal behandles. Det er tilfældet hvis det hidtil nærmeste punkt er mindst lige så langt væk fra p som afstanden fra p til den aktuelle knudes partitionsværdi (dvs. målt vinkelret ind på linien gennem partitionsværdien) — sammenligningerne i l. 10 og 14 i funktionen RNN. I fig. 6.2 illustreres søgning efter NN$(7, 5)$. Givet roden af træet vælges først det venstre barn. Til højre i figuren ses hvilken rute søgningen tager. Ved bladet med punkt $(9, 12)$ dannes første afstand, og i figuren til venstre ses at cirklen C_1 centreret i $(7, 5)$ med radius lig afstanden fra $(7, 5)$ til $(9, 12)$ overlapper regionen markeret ved $x <= 9, y <= 5$. Når kontrol returnerer fra bladet $(9, 12)$ ses at den kvadrerede afstand fra p til $(9, 12)$ ikke er mindre end den kvadrerede afstand langs

y-aksen fra 5 til 5. Altså skal det højre barn også behandles når der returneres fra behandling af bladet med punkt $(9, 12)$. Tilsvarende overlapper C_2 regionen til højre for $x = 9$ og derfor skal rodens højre barn også behandles. Dette sker når der returneres fra bladet indeholdende punktet $(3, 4)$.



Figur 6.2: *Figuren illustrerer søgning efter NN*$(7, 5)$*. Til venstre angiver cirklerne den hidtil korteste afstand. Til højre markerer bogstavernes rækkefølge hvilken rute søgningen foretager i træet. I 'b' udregnes første afstand, hvilket giver anledning til cirkel* C_1*. I 'e' udregnes anden afstand som giver anledning til cirkel* C_2*. I 'g' udregnes tredie afstand, som er for stor og derfor ignoreres. Næste blad finder det rigtige svar, punktet* $(10, 4)$*.*

Koden for ANN-søgning i Kd-træer består af to funktioner, NN(p) og RNN(k).

```
NN(punkt p)
1  nntarget ← p
2  nndist ← ∞
3  nn = NULL
4  RNN(rodknude)

RNN(knude k)
 1  if (k er et blad)
 2      a ← kvadreret afstand fra nntarget til k
 3      if (nndist > a)
 4              nndist ← a
 5              nn = k.bladdata
 6  else
 7      diff ← afstand fra nntarget til k.v langs en akse
 8      if (diff < 0)
 9              RNN(k.venstrebarn)
10              if (nndist ≥ diff²)
11                  RNN(k.højrebarn)
12      else
13              RNN(k.højrebarn)
14              if (nndist ≥ diff²)
15                  RNN(k.venstrebarn)
```

I RNN linie 7 udregnes *diff* som afstand langs en akse, hvilket blot betyder at man i 2 dimensioner kan trække $k.v$ fra p's x- eller y-koordinat, afhængig af

k.d. Resultatet *diff* er negativt eller 0 hvis p ligger til venstre for/under denne interne knudes skillelinie.

6.2 Kompleksitet

Med min implementering tager konstruktion af et Kd-træ for n K-dimensionelle punkter i værste fald $O(Kn^2)$ tid. I hvert kald til DAN-KDTRÆ hvor den aktuelle delmængde af S indeholder mere end et element er der to opgaver: 1) Find dimension med størst spredning, og 2) find medianen i den valgte dimension. Første opgave tager $O(Kn)$ tid for n punkter. Medianfinder-algoritmen jeg anvender er den simpleste af de to beskrevet i [13] og tager forventet lineær men i værste fald kvadratisk tid. Jeg anvender ikke pegere, så der skal flyttes K elementer for hvert punkt, og den præcise tidskompleksitet for medianfinder-algoritmen er derfor $O(Kn^2)$ i værste fald. Algoritmens værstefalds-køretid kan beskrives ved denne rekursion:

$$T(n) = \begin{cases} K & \text{for } n = 1, \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(Kn^2) & \text{for } n > 1. \end{cases}$$

Det vises herunder med induktion at $T(n) \leq cKn^2$.

Bevis. Som induktionsbasis anvendes dette, med $c \geq 2$:

$$\begin{aligned} T(1) = K &\leq cK, \\ T(2) = 6K &\leq 4cK. \end{aligned}$$

Herunder ses induktionsskridtet.

$$\begin{aligned} T(n) &\leq cK(\lceil \frac{n}{2} \rceil^2) + cK(\lfloor \frac{n}{2} \rfloor^2) + Kn^2 \\ &\leq cK((\frac{n+1}{2})^2 + (\frac{n}{2})^2) + Kn^2 \\ &= cK(\frac{n^2+2n+1}{2^2} + \frac{n^2}{2^2}) + Kn^2 \\ &= cK(2\frac{n^2}{2^2} + \frac{2n}{2^2} + \frac{1}{4}) + Kn^2 \\ &= \frac{c}{2}Kn^2 + \frac{c}{2}Kn + \frac{c}{4}K + Kn^2 \\ &= (\frac{c}{2} + 1)Kn^2 + \frac{c}{2}Kn + \frac{c}{4}K. \end{aligned}$$

Vælges $c = 6$ får man dette:

$$\begin{aligned} 4Kn^2 + 3Kn + \frac{3}{2}K &\leq 6Kn^2 \Leftrightarrow \\ 3n + \frac{3}{2} &\leq 2n^2, \end{aligned}$$

som holder for $n \geq 2$. □

Der er dog også i [13] en median-finder med værste-falds køretid på $O(n)$. Hvis den bruges, og man også benytter pegere, vil konstruktion af kd-træet tage $O(Kn \log n)$ for n punkter i K dimensioner. Rekursionen er som følger:

$$T(n) = \begin{cases} 1 & \text{for } n = 1, \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(Kn) & \text{for } n > 1. \end{cases}$$

Det vises her med induktion at $T(n) \leq cKn \log n$.

Bevis. Basis for induktionen er $T(2)$ og $T(3)$:

$$\begin{aligned} T(2) = T(1) + T(1) + 2K = 2 + 2K &\leq 2\log(2)cK = 2cK. \\ T(3) = T(2) + T(1) + 3K = 3 + 5K &\leq 3\log(3)cK \approx 4.75cK. \end{aligned}$$

For $c \geq 2$ holder ovenstående. Induktionsskridtet er herunder:

$$\begin{aligned} T(n) &\leq cK\lceil \frac{n}{2} \rceil \log\lceil \frac{n}{2} \rceil + cK\lfloor \frac{n}{2} \rfloor \log\lfloor \frac{n}{2} \rfloor + Kn \\ &\leq cK(\frac{n+1}{2} \log \frac{n+1}{2} + \frac{n}{2} \log \frac{n}{2}) + Kn \\ &= cK(\frac{n+1}{2} \log(n+1) - \frac{n+1}{2} + \frac{n}{2} \log(n) - \frac{n}{2}) + Kn \\ &= cK(\frac{n+1}{2} \log(n+1) + \frac{n}{2} \log(n) - n - \frac{1}{2}) + Kn. \end{aligned}$$

Spørgsmålet er hvornår dette gælder:

$$\begin{aligned} cK(\frac{n+1}{2} \log(n+1) + \frac{n}{2} \log(n) - n - \frac{1}{2}) + Kn &\leq cKn \log n \Leftrightarrow \\ c(\frac{n+1}{2} \log(n+1) + \frac{n}{2} \log(n) - n - \frac{1}{2}) + n &\leq cn \log n. \end{aligned}$$

For $c = 10$ får man dette:

$$\begin{aligned} (5n+5)\log(n+1) + 5n\log(n) - 10n - 5 + n &\leq 10n\log(n) \Leftrightarrow \\ (n+1)\log(n+1) + n\log(n) - \frac{9}{5}n - 1 &\leq 2n\log(n) \Leftrightarrow \\ (n+1)\log(n+1) - n\log(n) - \frac{9}{5}n - 1 &\leq 0 \Leftrightarrow \\ n\log(n+1) + \log(n+1) - n\log(n) - \frac{9}{5}n - 1 &\leq 0. \end{aligned} \tag{6.1}$$

På grund af følgende:

$$\begin{aligned} \log(n+1) - \log(n) &\leq 1 \Rightarrow \\ n(\log(n+1) - \log(n)) &\leq n \Rightarrow \\ n(\log(n+1) - \log(n)) - n &\leq 0, \end{aligned}$$

kan (6.1) reduceres til:

$$log(n+1) - \frac{4}{5}n - 1 \leq 0,$$

som holder for alle $n \in \mathbb{N}$. □

ANN-søgning har iflg. [15] en værste-falds-tid på $O(n \log^K n)$ for n punkter i K dimensioner hvis dimensionen der opdeles i er den der har størst spredning, dvs. som i min implementering.

6.3 Bladstørrelse

Jeg har kun set på Kd-træer med blade indeholdende et punkt. Det er relativt simpelt at øge antal punkter i et blad: man skal, udover at benytte en udvidet struktur (fx en liste) i bladet, i søgninger efter NN(p) udregne afstanden fra p til samtlige punkter i bladet. I [44] rapporteres at NN-søgninger i to dimensioner er hurtigst med to punkter pr. blad og marginelt langsommere med et punkt pr. blad. Derefter stiger søgetid med antal punkter/blad. For højere dimensioner ses parabel-lignende kurver for søgetiden, hvor minimum ligger mellem 8 og 16 punkter/blad. Disse søgninger er i det der kaldes *signal data*. Denne type data benyttes for at vise fordelen ved at opdeling ikke nødvendigvis foregår ved linier der er vinkelrette på koordinatsystemets akser. Min implementering opdeler dog efter linier der *er* vinkelrette på akserne, men jeg antager at tallene stadig er relevante.

6.4 Numerisk robusthed

Algoritmen til ANN-søgning i Kd-træer er som man kan se ganske begrænset i omfang. Den er også robust, bortset fra at overløb kan forekomme. I konstruktion af træet foregår der intet af numerisk interesse når et blad oprettes; oprettelse af indre knuder medfører udregning af hvilken dimension der skal opdeles på, her sker kun subtraktion og sammenligning, og udregning af median, hvor der kun sker sammenligninger. Der er altså ikke behov for at ændre på algoritmen for at gøre den robust. I ANN-søgning forekommer multiplikation og subtraktion i afstandsberegningerne, så overløb kan forekomme, men ellers sker der kun sammenligninger. Heller ikke her er der altså noget at gøre. Hvis overløb skal håndteres er der flere muligheder. Hvis man ikke kan skalere inddata er typer med ubegrænset eller multi-præcision (fx. [20] eller [2]), er en mulighed, men gør som nævnt tidligere oftest algoritmen en (hel) del langsommere.

7

Implementering

Her beskrives de eksperimenter jeg har lavet med algoritmer og metoder som blev undersøgt i de forrige kapitler. Først kort om programmerne, derpå de overordnede resultater, så detailresultater og til sidst afsluttende bemærkninger.

7.1 Programmerne

Programmerne, alle skrevet i C++, er to versioner af plane sweep-algoritmen, tre versioner af Delaunaytriangulering med CGAL, tre versioner af Delaunaytriangulering med TTL, en version af Kd-træet og en brute force-udgave. De bruger alle, med to undtagelser, double som type for punktkoordinater. De to undtagelser er plane sweep med rationelle typer og CGAL GMP, som benytter hhv. LEDA::rational og GMP, begge multipræcisionstyper. Blandt dem der bruger double som type kræver nogle heltalligt inddata. Her gemmes data dog også i double-typer.

Programmernes korrekthed er testet ved hjælp af brute force-versionen. Det foregår ved først at lade et program udregne ANN for en punktmængde, og derpå lade brute force-programmet udregne ANN for samme punktmængde. En sammenligning afgør derefter om det første resultat stemmer overens med det andet. I ingen af de kørsler jeg foretog på denne måde blev der fundet uoverensstemmelser, hvilket vil sige at de to programmer er enige om resultatet, ikke at det er garanteret korrekt. Brute force-versionen gør dette: For en punktmængde $S \subset \mathbb{R}^2$ med n punkter sammenlignes for hvert punkt p afstanden fra p til de $n-1$ andre. Det nærmeste blandt de $n-1$ andre vælges som p's nærmeste nabo. Algoritmen tager $\Theta(n^2)$ tid. Kontrol af 200.000 punkter med brute force tager ca. 4 timer. Det er korrekt at brute force-programmet også kan rammes af afrundingsfejl. For at undgå dette ville man skulle benytte multipræcisionstyper. Dette ville gøre brute force virkelig langsomt, så det er ikke sket. Jeg mener at robusthed godt kan undersøges på trods af dette.

Visse af programmerne stiller krav til koordinaterne, enten en øvre grænse for hvor store de må være eller at det skal være heltal. Herunder ses en skematisk fremstilling af programmerne og evt. krav til inddata. Som sagt benytter alle programmerne udover Plane sweep 2 og CGAL GMP, double som basal

kordinattype, også når der kræves heltalligt inddata.

Navn	Robust?	Krav til inddata	Grænse?
Plane sweep 1	Nej	Heltal	-
Plane sweep 2	Ja, LEDA::rational	Heltal	-
CGAL Ikke-robust	Nej	Ingen	-
CGAL Filter	Ja, filter	Ingen	-
CGAL GMP	Ja, GMPq	Heltal	-
TTL Ikke-robust	Nej	Ingen	-
TTL Filter 1	Ja, filter (Shewchuk)	Ingen	eksp. $\in [-142, 201]$
TTL Filter 2	Ja, filter (ABDPY)	Heltal	$\pm 2^{51}$
Kd-træ	Ja	Ingen	-

Tabel 7.1: Tabel over de programmer jeg har lavet.

7.2 Data og computer

Som inddata benytter jeg denne (primære) serie af punktmængder,

$$200.000, 400.000, 800.000, 1.600.000, 3.200.000,$$

og derudover tre mindre mængder på ca. 32898 punkter. Disse tre mindre mængder består af flydende tal som alle ligger næsten på samme linie. De er med i et forsøg på at presse filtrene mest muligt, men kan desværre ikke bruges til de fire programmer der kræver heltal. For de to programmer der anvender multipræcisionsmetoder køres kun de tre første primære mængder, da man allerede der tydeligt ser tidsforbruget, som er meget stort. Det var oprindeligt tanken også at have en mængde med 6.4mill. punkter, men der er problemer for to af programmerne for større datamængder, nærmere bestemt har mit plane sweep program en hukommelseslækage, og TTL-biblioteket går i stå (overdreven brug af *swap memory*). For begge gælder at de kan klare 3.2mill punkter, men altså ikke 6.4mill.

I den primære serie anvendes to forskellige fordelinger af punkter for de programmer der ikke kræver heltal: Den ene er tilfældige punkter i enhedskvadratet og den anden er punkter der ligger omtrentligt på enhedscirklen. Den sidste mængde er inspireret af [41], som nævner at den er spcielt hård for trianguleringsprogrammer. Det har som ventet haft en vis effekt på trianguleringsprogrammerne. Det er dog ikke trivielt at finde data der virkelig sætter et program på prøve, og for de programmer (to plane sweep-programmer, CGAL GMP og TTL ABDPY) der kræver heltal har jeg kun brugt en type inddata. Data er opdelt i disse typer, som jeg for nemheds skyld refererer til i graferne (typerne 1 og 3 kom af forskellige årsager ikke med):

2 Tilfældige flydende tal i enhedskvadratet.

4 Flydende tal omtrentligt på enhedscirklen.

5 Tilfældige heltal i intervallet $[0, 2^{31}]$. Disse tal gemmes dog i doubles.

6 Tilfældige heltal i intervallet $[-2^{51}, 2^{51}]$. Disse tal gemmes i doubles.

Alle programmer er kørt på en AMD Sempron 3000+, Linux 2.6.22 (Ubuntu 7.10) med 1GB hukommelse. Ingen andre brugerprogrammer kørte mens mine programmer kørte. Hvert program blev kørt fire gange, bortset fra de tre mindre punktmængder som blev kørt ti gange hver, og resultatet jeg viser er gennemsnittet.

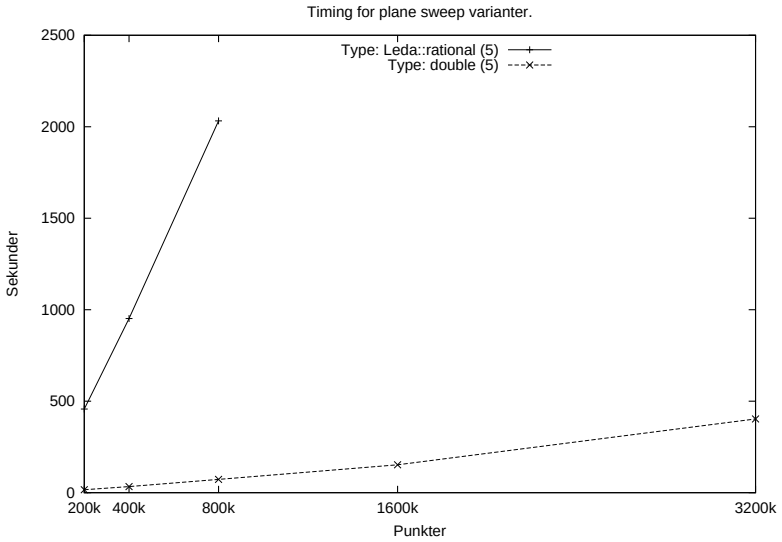
7.3 Resultater

Herunder kommer beskrivelser af de forskellige resultater. Først beskrives de overordnede resultater, dvs. hvor lang tid programmerne har brugt på at løse hele opgaven ANN (dvs. inklusiv evt. konstruktion af triangulering eller Kd-træ), og bagefter beskrives kort detaljer for de algoritmer der først konstruerer en datastruktur og derpå løser ANN. I alle diagrammer er linier mellem punkter kun til stede for at hjælpe visualiseringen.

7.3.1 Overordnede resultater

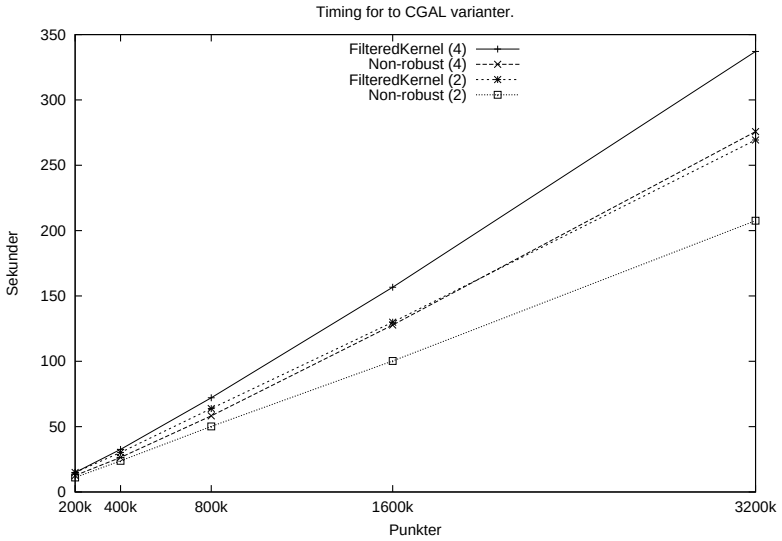
Plane sweep

Mit plane sweep program er som forventet klart hurtigere med den indbyggede double-type end med LEDA::rational. Iflg. LEDA-dokumentation [2] er denne type 30-100 gange langsommere end den indbyggede double. Mit program ligger altså i den lave ende — de tre målinger ligger alle omkring en faktor 30 over de tilsvarende for programmet der anvender double.



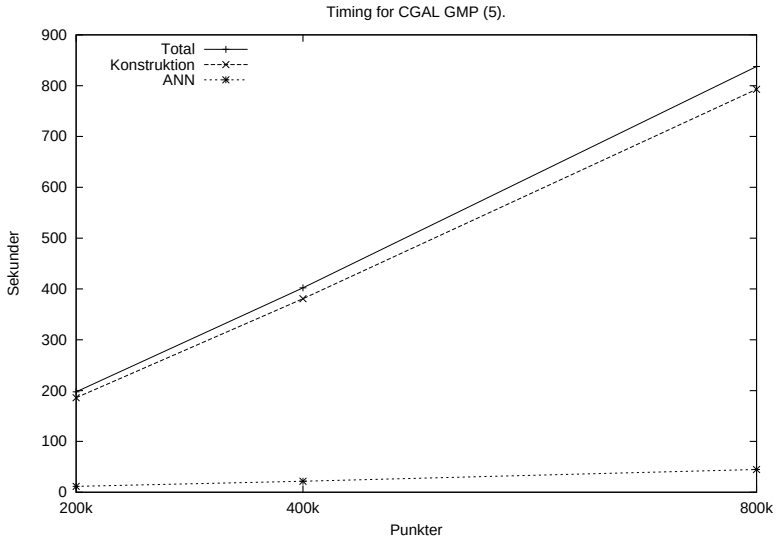
Triangulering: CGAL ikke-robust & filtreret

Her vises to CGAL-versioner, ikke-robust og filtreret (begge bruger Delaunayhierarki til punktlokalisering). Begge er lidt hurtigere end den ikke-robuste plane sweep-version. CGAL-programmet påvirkes tydeligt af at skifte fra almindelig, ikke-robust kerne til filtreret kerne. For den største punktmængde, 3200k, tager den robuste version godt 30% længere tid end den ikke-robuste. Som man ser er inddatatype 4 også hårdere for programmet end type 2. Det er som forventet for trianguleringsprogrammer.



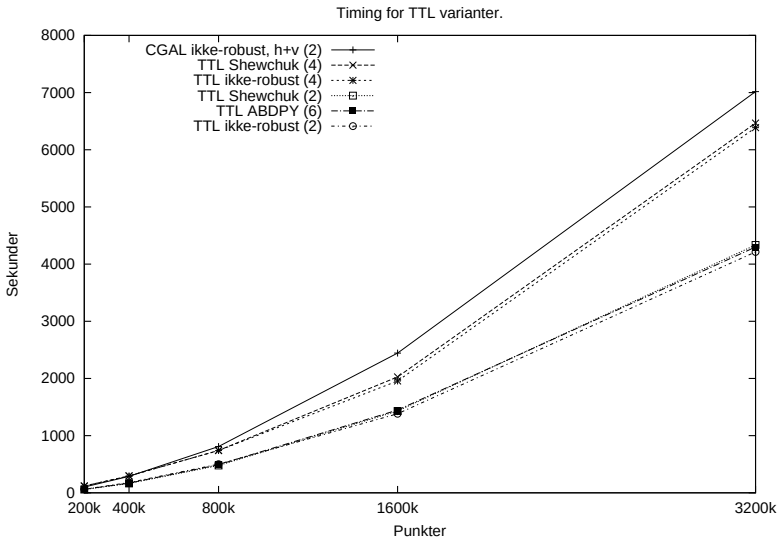
Triangulering: CGAL GMP

Den version af CGAL-trianguleringsprogrammet der anvender GMP (og Delaunayhierarki til punktlokalisering) er også helt som ventet meget langsommere, og vises her i en separat graf. Denne graf viser en kurve for konstruktion af trianguleringen (i midten) og en tredie (nederst) for ANN-delen af algoritmen. Den øverste kurve er summen af de to nedre, dvs. den totale køretid for programmet. Bemærk også at jeg kun har kørt denne version for 200k, 400k og 800k punkter.



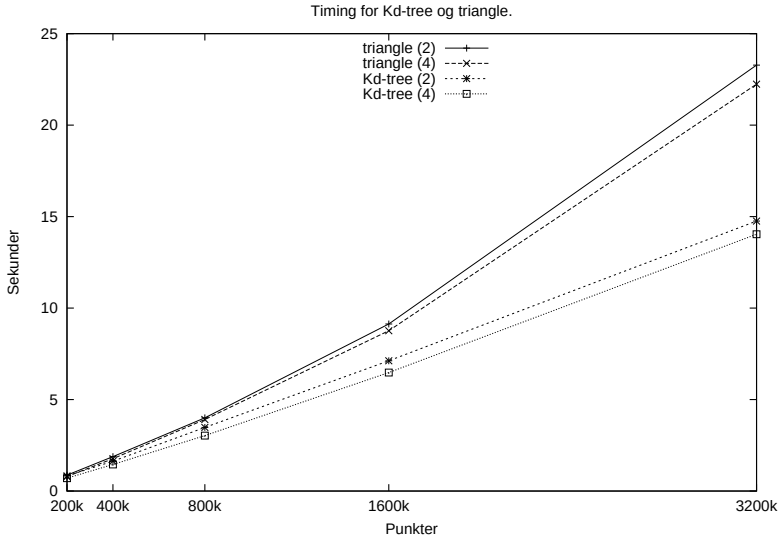
Triangulering: TTL

TTL viser sig at være langsom sammenlignet med CGAL. Det hævdes at punktlokalisering er flaskehalsen i inkrementel Delaunaytriangulering. For at undersøge det kørte jeg CGAL ikke-robust med hop+vandring til punktlokalisering. Resultatet ses som den øverste kurve i grafen herunder, og viser at med hop+vandring til punktlokalisering stiger køretiden for dette CGAL-program med en faktor 20. De resterende kurver kan opdeles i to grupper, de to umiddelbart under CGAL-kurven er TTL kørt på type 4 data, mens de nederste tre kurver er TTL kørt på type 2 og 6. CGAL-kurven bør sammenlignes med den nederste, dvs. for TTL ikke-robust med data af type 2. At der ses samme køretid for robuste og ikke-robuste versioner skyldes formentlig, i hvert fald for Shewchuks prædikater, at inddata ikke presser filtrene tilstrækkeligt. De ender med i realiteten kun at udføre flydende tals-beregning. Det er dog overraskende at tidsforbrug er det samme også for data af type 4. Hvad ABDPY-prædikaterne angår antager jeg at det samme gælder. Disse prædikater har jeg dog ikke analyseret indgående. TTL viser sig at være følsom overfor inddata i større omfang end CGAL med Delaunayhierarki. Tidsforbruget stiger med op til 50% ved skift fra type 2 til 4, også for den ikke-robuste version. Det kan muligvis igen tilskrives den punktlokalisering TTL anvender.



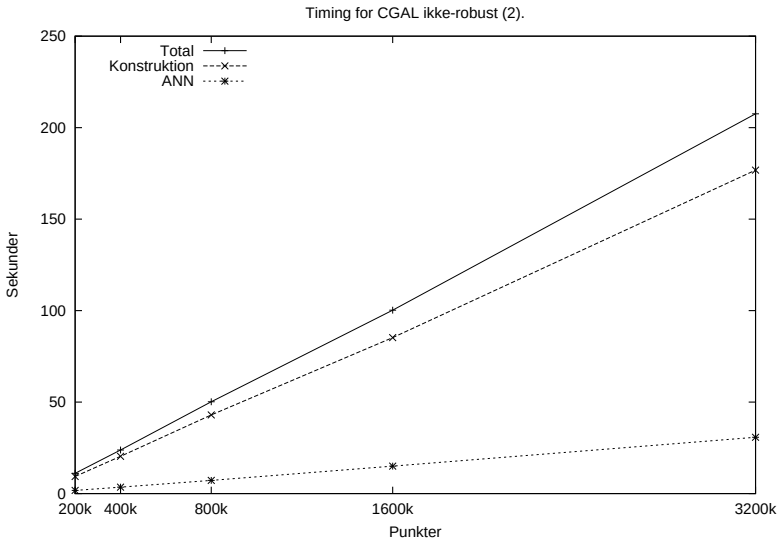
Kd-træer og *triangle*

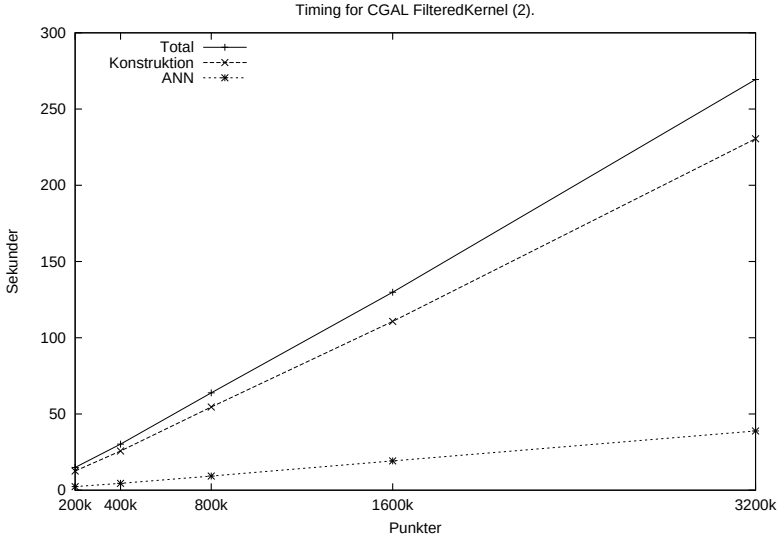
Som man ser på grafen herunder er Kd-træer ikke blot robuste i sig selv, de er også langt de hurtigste af de algoritmer jeg ser på. Her ses i øvrigt at det der var tænkt som hårde inddata til trianguleringsalgoritmer, bearbejdes hurtigere med Kd-træet end punkter i enhedskvadratet. Tidsforbrug for Kd-træet var så overraskende at jeg til sammenligning kørte Shewchuks *triangle*-program [42], der danner en Delaunaytriangulering med del-og-hersk-algoritmen [25] men som ikke løser ANN. Dette program anvender også de adaptive prædikater som jeg kalder Shewchuk-prædikater. Det viser at det ikke er urealistisk når konstruktion af Kd-træ og beregning af ANN tager så kort tid som det gør.

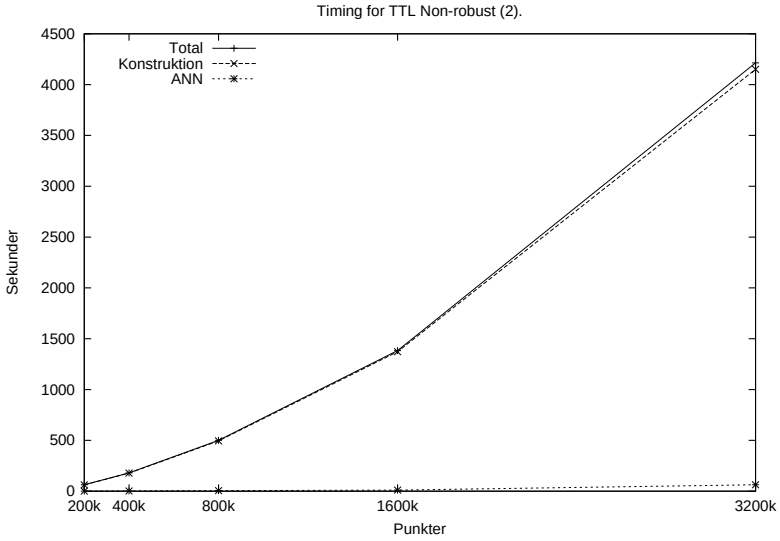


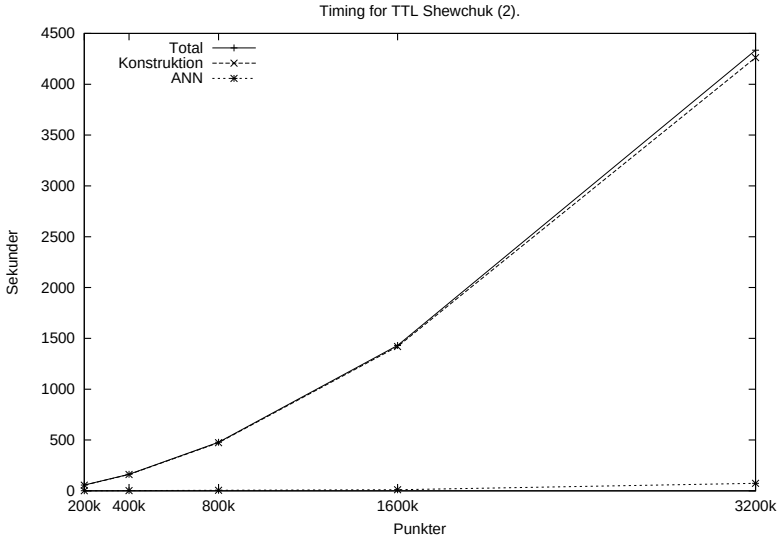
7.3.2 Detailresultater

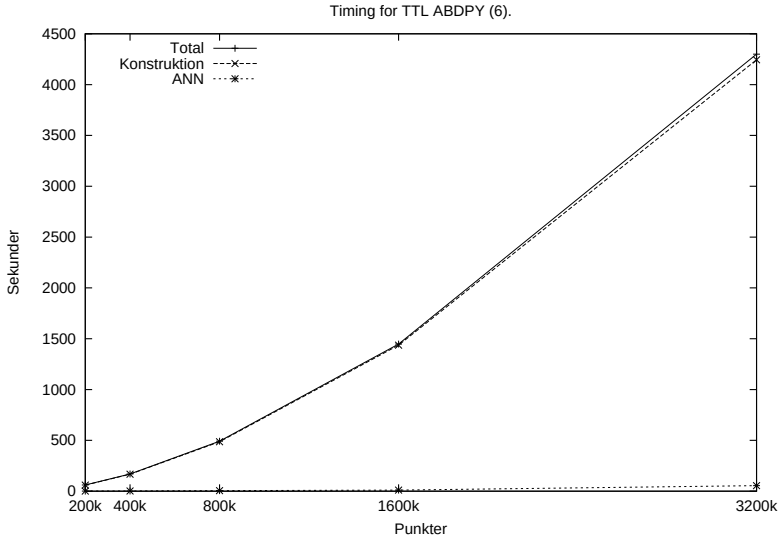
Herunder vises grafer med køretid opdelt i konstruktion af triangulering/Kd-træ og ANN-søgning. Der er ingen forskel i forhold mellem tidsforbrug til konstruktion og ANN-beregning ved skift fra den ene type data til den anden, så kun kørsler med data af en type, enten 2 eller 6, vises her.

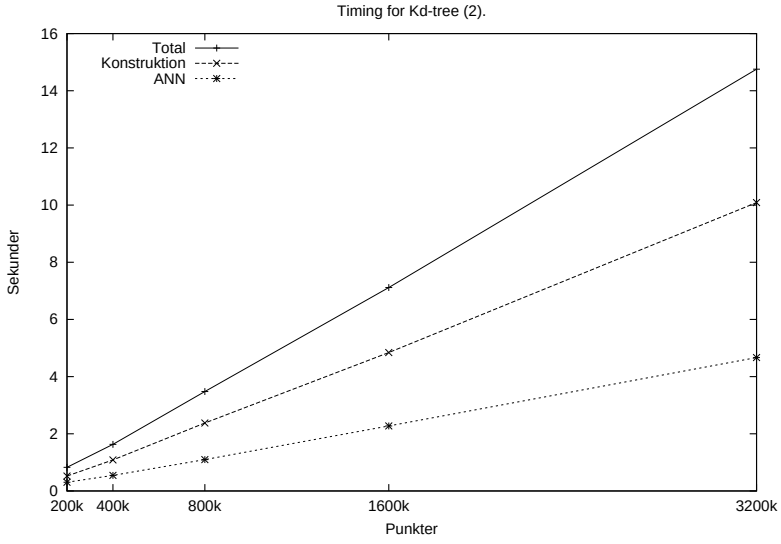





Timing for TTL Non-robust (2).
Total
Konstruktion
ANN
Sekunder
Punkter
0
500
1000
1500
2000
2500
3000
3500
4000
4500
200k
400k
800k
1600k
3200k


Timing for TTL Shewchuk (2).
Total
Konstruktion
ANN
Sekunder
Punkter
0
500
1000
1500
2000
2500
3000
3500
4000
4500
200k
400k
800k
1600k
3200k





Det var ventet at ANN-delen ville udgøre en lille del af den samlede køretid for trianguleringsalgoritmerne, og en større del mht. Kd-træet. En triangulering af n punkter har som nævnt færre end $3n$ kanter, mens konstruktion af trianguleringen tager $O(n \log n)$. For Kd-træet tager konstruktion i dette tilfælde formelt set $O(Kn^2)$, men den anvendte medianfinder er i praksis meget hurtig.

7.3.3 Tre mindre punktmængder

To af de tre mindre punktmængder er inspireret af [32], og som sagt brugt i et forsøg på at presse filtrene maksimalt. I [32] beskrives bl.a. hvordan orient2d i ikke-robust version udviser hvad de kalder en mærkelig geometri[1]. De tester prædikatet på tre punkter i planen, hvoraf to ligger eksakt på og et næsten på diagonalen $x = y$. Resultatet viser at orient2d med flydende tal i visse tilfælde svarer forkert. Jeg har dannet to punktmængder ud fra denne ide: Den ene, kaldet KMPSY, har to punkter på diagonalen $x = y$, og 32896 andre punkter som er perturberet en smule, mens jeg i den anden, kaldet KMPSY-2, blot ikke medtager de to punkter $(12.0, 12.0)$ og $(24.0, 24.0)$. Punkterne dannes nærmere bestemt på følgende måde (for KMPSY-2 er linie 2 og 3 fjernet).

DAN-KMPSY-PUNKTER:

```
1  vector<Point<double> > pointset;
2  pointset.push_back(Point(12.0, 12.0));
3  pointset.push_back(Point(24.0, 24.0));
4  double eps = pow(2.0, -53);
5  for (unsigned  int x=0; x<256; x++)
6      for (unsigned int y=0; y<=x; y++)
7              pointset.push_back(Point(0.5+x*eps, 0.5+y*eps));
```

Punkterne i KMPSY-2 ligger helt regelmæssigt i et grid, mens de to punkter med 'store' koordinater i KMPSY gør at der her vil komme 'skæve' kanter i trianguleringen. Resultatet for mine programmer med disse punktmængder, indeholdende 32898 (og 32896) punkter, og en punktmængde med 32898 tilfældige punkter i enhedskvadratet ses herunder. Tallene er i sekunder og er samlet tid for konstruktion af triangulering/Kd-træ og ANN-beregning, nøjagtigt som i de overordnede resultater i afsn. 7.3.1. Tallene er gennemsnittet af ti kørsler.

Program	Tilfældige punkter i enhedskvadratet	KMPSY	KMPSY-2
TTL Ikke-robust	4.11	24.10	26.96
TTL Filter (Shewchuk)	3.80	21.98	22.69
CGAL Ikke-robust	1.81	Fejler	13.86
CGAL Filter	2.29	39.64	16.03
Kd-træ	0.12	0.25	0.22
triangle	1.10	1.43	1.34

Tabel 7.2: Tidsforbrug i sekunder ved ANN-beregning for tre mindre punktmængder.

Tallene viser et blandet mønster. For det enkelte program stiger tidsforbruget voldsomt, bortset fra de to sidste, når man går fra tilfældige punkter til både KMPSY og KMPSY-2. Dette kan for alle TTL- og CGAL-trianguleringsprogrammer forklares med et større antal kald til prædikaterne og at en del af disse kald går videre end til første trin. For de tilfældige punkter er der aldrig brug

[1] Se `http://www.mpi-inf.mpg.de/departments/d1/ClassroomExamples/failure_a1.html`.

for andet end første trin i filtret, dvs. flydende tal-beregning. Kd-træet bruger dobbelt så meget tid på KMPSY og KMPSY-2 som på tilfældige punkter. Jeg har ikke nogen forklaring på dette, koordinater i alle tre mængder er af samme præcision og størrelsesorden. Det gælder i øvrigt for alle TTL- og CGAL-trianguleringsprogrammer at det øgede tidsforbrug udelukkende kommer fra konstruktion af trianguleringen, mens ANN-beregning tager ca. samme tid med alle tre punktmængder, hvilket ikke er overraskende. Det er dog overraskende at TTL Filter (Shewchuk) med KMPSY og KMPSY-2 faktisk er hurtigere end TTL Ikke-robust. Sidste linie viser *triangle*-programmet [42]. Det anvender som sagt en anden trianguleringsalgoritme, så sammenligning er svær, men i hvert fald ses en meget mindre stigning i køretid for *triangle* ved skift fra tilfældige punkter til KMPSY end for både TTL Filter (Shewchuk) og CGAL Filter.

I tre kørsler af TTL Filter (Shewchuk) for hhv. tilfældige punkter, KMPSY og KMPSY-2 målte jeg antal kald til de forskellige trin i orient2d- og incircle-filtret (trin D er eksakt beregning). Tallene ses herunder.

Prædikat	Tilfældige punkter i enhedskvadratet	KMPSY	KMPSY-2
orient2d A	2961298	98389	72713
orient2d B	0	574	1225
orient2d C	0	510	1223
orient2d D	0	2	65
incircle A	510856	5622041	5617723
incircle B	0	33115	33989
incircle C	0	759	521
incircle D	0	255	0

Tabel 7.3: Antal kald til orient2d og incircle i TTL Filter (Shewchuk).

Antal kald til incircle ti-dobles for KMPSY og KMPSY-2, og en relativt lille del af disse kald går videre end til trin A. Det er ikke uventet, men det er overraskende at antal kald til orient2d falder drastisk ved skift fra tilfældige punkter til KMPSY og KMPSY-2. At der med tilfældige punkter kun behøves trin A vidner om at IEEE 754 er et godt system når data er pænt fordelt som her.

Antal kald til de to prædikater for CGAL Filter ses herunder.

Prædikat	Tilfældige punkter i enhedskvadratet	KMPSY	KMPSY-2
orient2d A	1836436	8134952	10795812
orient2d B	0	485306	1358
incircle A	304512	5605748	5540437
incircle B	0	97706	32426

Tabel 7.4: Antal kald til orient2d og incircle i CGAL Filter.

For CGAL sås i tabel 7.2 at CGAL Ikke-robust fejler med KMPSY-mængden. Det sker via en assertion — `terminate called after throwing an instance`

`of 'std::logic_error'` — under punktlokalisering. At programmet ikke fejler med KMPSY-2 kan jeg ikke forklare. At CGAL Filter er meget hurtigere til KMPSY-2 end KMPSY skyldes formentlig primært at meget færre kald til orient2d fejler i trin A. Den voldsomme stigning i køretid for CGAL Filter ved skift fra tilfældige til KMPSY-punkter skyldes at prædikaterne for det første kaldes mange flere gange og for det andet at et stort antal af disse kald går videre end til trin A. Bemærk at punktlokalisering sker med Delaunay-hierarki, så man kan ikke sammenligne TTL og denne CGAL-version på det punkt. Det var overraskende at CGAL Filter bruger hele 80% længere end TTL Filter (Shewchuk). I [36] findes en måling der viser at CGAL med et filter bestående af tre trin (som beskrevet i 4.3.3) er hurtigere end Shewchuks prædikater. Det nævnes dog også at det er for en punktmængde med tilfældige tal, der *"hardly triggers a robustness failure"* (afsnit *B: Benchmarks*). Det er fristende at konkludere at Shewchuks filter er hurtigere end det der benyttes i CGAL Filter. Jeg ved dog ikke om det er rimeligt, i og med at TTL åbenlyst er et langsommere bibliotek til Delaunaytriangulering. Der er dog ikke tvivl om at CGAL filtret, for KMPSY-punkterne, i absolutte tal meget oftere fejler i første trin og må benytte det dyrere trin B. Jeg har som sagt ud fra artikler sluttet mig til at CGAL filtret anvender tre trin. Hvis det viser sig at være forkert, og at man allerede i trin B benytter multipræcisionstyper vil det kunne forklare den lange køretid for CGAL Filter med KMPSY-punkterne.

Overordnet set er resultaterne dog som ventet, nemlig at robusthed har en tydelig pris, at de omtalte filtre er multipræcisionstyper overlegne og at flydende tal-beregning er tilstrækkeligt i mange situationer.

8

Konklusion

Ved start af dette projekt udarbejdede jeg en række projektspecifikke læringsmål, og det virker rimeligt at trække dem frem på dette sted.

Læringsmål for specialet

1. At analysere og forstå problemet med præcision
2. At forstå løsningerne
3. At kunne vurdere kvaliteten af en given metode i en given sammenhæng
4. At kunne implementere en løsning
5. At kunne vurdere hvornår man risikerer problemer/fejl pga. dette problem

De første fire mål synes jeg er nået, mens det sidste, som også kunne formuleres som et ønske om at beskrive typiske robusthedsfælder, kun delvist er nået. For at den sætning skal være sand vil jeg her pege på en række konkrete steder hvor problemer kan opstå: Beregning af determinanter, kvadratrodsberegning (rodberegninger generelt, omend det måske ikke ses så ofte i algoritmisk geometrisk sammenhæng — se fx [27]). Afstand kan i mange tilfælde beregnes som kvadreret afstand. Dertil kommer at konstruktion af nye geometriske objekter medfører risiko for akkumulerede fejl. Som vist er det dog ikke helt nemt at fremprovokere fejl.

Hvad de første fire læringsmål angår synes jeg som sagt de er nået. Jeg har fået indsigt i IEEE 754, i robusthedsproblemer i geometriske algoritmer og løsninger dertil, primært eksakte typer og filtre til flydende tal. Jeg har analyseret forskellige geometriske algoritmer der løser problemet alle nærmeste naboer, jeg har analyseret et antal mulige løsninger på problemet og implementeret en række algoritmer i både robuste og ikke-robuste versioner. Til sidst har jeg eksperimenteret med disse programmer, og har i praksis set konsekvenserne af at indbygge forskellige robuste metoder i geometriske programmer.

Videre arbejde

I umiddelbar forlængelse af dette projekt kunne det være interesant at undersøge det store fald i kald til orient2d i TTL, at undersøge CGAL filtret nærmere

og at finde ud af hvorfor CGAL fejler på en bestemt punktmængde.

Derudover ville jeg kunne bruge tid på følgende:

- At udregne et mål for hvor mange instruktioner en given filtermetode bruger (på hvert trin). Det kunne bruges i sammenligninger.
- At undersøge Kd-træer i flere dimensioner og hvordan range-søgninger i Kd-træer påvirkes af måden diskriminatoren udvælges på.
- At undersøge multipræcisionstyper, bl.a. forskellene mellem multi-ciffer og multi-komponent-metoder, bl.a. i forbindelse med brug af meget store heltal.
- At studere de topologiske metoder beskrevet i [47, 46].

Bilag A

Implementeringsdetaljer

A.1 TTL triangulering

TTL består af et bibliotek, som tager sig af selve Delaunaytrianguleringen. Et niveau over den findes en anden klasse kaldet Triangulation, som er den der implementerer selve strukturen for trianguleringen (dvs. halvkanter etc.). Øverst findes så det program, `main.cpp` der samler det hele. Som det er nu afgøres det hvilken type prædikat der anvendes i klassen Triangulation. Hvis man skal skifte prædikat-type, fx fra robust til ikke-robust, skal biblioteket altså genkompileres. Beslutningen om hvilke prædikater der anvendes burde ligge i `main.cpp`, og det kunne ske ved at gøre prædikaterne til en template-parameter for Triangulation-klassen,

A.2 Plane-sweep

A.2.1 Datastrukturer

I det følgende beskrives, i kort form, detaljer vedrørende implementering af *plane sweep*-algoritmen.

For nemheds skyld behandles de to første punkter manuelt, dvs. de tages ud af køen, sættes som hinandens NN_v, og bisektoren mellem dem indsættes i søgetræet.

Jeg benytter prioritetskøen fra C++ standardbiblioteket, se [45]. I denne kø anvender jeg to sammenligningsfunktioner, begge med stigende x-koordinater og med skæringspunkter som 'mindre end' datapunkter. Forskellen er, at den ene sorterer y-koordinater stigende, mens den anden sorterer y-koordinater faldende. Dette fanger situationer hvor fx tre punkter har samme x-koordinat og tre forskellige y-koordinater, og det øverste er det mellemstes nærmeste nabo.

Som søgetræ anvendes *map*, også en del af C++ standardbiblioteket. En map er en såkaldt *sorteret associativ beholder*. Ved søgning efter et punkt p's interval i søgetræet T dannes en vandret bisektor gennem p, og derpå søges efter

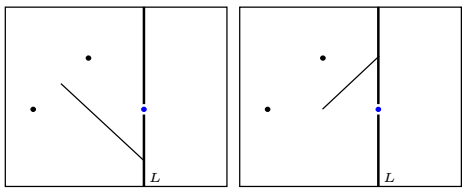
det mindste element i T, hvis nøgle er større end p's ($upper\_bound$).

Resultatet af hver af de to første faser gemmes i en *map*. For hvert punkt er der altså to forslag til hvad der skal være det endelige resultat. For endeligt at afgøre dette, sammenlignes den kvadrerede afstand fra punktet til hvert af de to forslag.

Hvis et punkt p og et punkt q har samme koordinater, dvs. de er identiske, har jeg bestemt at de er hinandens nærmeste naboer. Der kunne tænkes situationer hvor det var nyttigt at et punkts nærmeste nabo ikke må have samme koordinater som punktet selv, men dem beskæftiger jeg mig ikke med her.

A.2.2 Algoritmisk robusthed

I en situation som i fig. A.1, hvor to punkter har samme y-koordinat og et tredie over og mellem dem, vil man, uden ekstra forholdsregler, få indsat både den skrå bisektor som ses i figuren til højre, og en lodret mellem de to nederste punkter. Dette er forkert, så derfor har jeg tilføjet en betingelse der udelukker lodrette bisektorer fra at blive indsat hvis man opdaterer nedad. I tilfælde af at en lodret bisektor virkelig skal indsættes, vil det ske i opdatering opad.



Figur A.1: *Til venstre situationen før det nye punkt er behandlet. Til højre ses det ønskede resultat, nemlig at de to punkter længst til højre er de eneste aktive.*

A.2.3 Typeskift i programmet

Her beskrives i punktform hvilke trin der skulle til for at konvertere mit plane-sweep-program fra at bruge *double* til at bruge *leda::rational*.

1. Intervalfunktioner bruger getYValueAsDouble, hvilket skal ændres.
2. ANNPlaneSweep: Skal *#include* Bisector_rat.h og intervalfunctions_rat.h.
3. ANNPlaneSweep: I updateYTable, efter while-loop, skal getYValueAsDouble ændres.
4. ANNPlaneSweep: I handleIntersection skal skæringspunkt ikke gemmes i en *double*.
5. ANNPlaneSweep: I calcResult skal afstand ikke gemmes i en *double*; flere steder.
6. Bisector: Skal *#include* leda::rational.

7. Bisector: Funktionen getYValueAsDouble skal returnere T, og have et andet navn.
8. Bisector: Alle kald til getYValueAsDouble skal ændres.
9. Bisector: I operatoren $<$ skal y-værdi udregnes anderledes (som angivet i artiklen).
10. Bisector: I sidste `else` skal `abs` og `min` skrives som hhv. `abs` og `leda::min`.

Litteratur

[1] Avnaim et al., Website: `http://www-sop.inria.fr/prisme/logiciel/determinant.html` (1995).

[2] Algorithmic Solutions, Website: `http://www.algorithmic-solutions.com/leda/index.htm` (2008).

[3] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec, Evaluating signs of determinants using single precision arithmetic, *Algorithmica* **17**, 2 (1997), 111–132.

[4] J. L. Bentley, Multidimensional binary search trees used for associative searching, *Commun. ACM* **18**, 9 (1975), 509–517.

[5] J. L. Bentley, K-d trees for semidynamic point sets, *SCG '90: Proceedings of the sixth annual symposium on computational geometry*, ACM (1990), 187–197.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry*, Second Revised Edition, Springer (2000).

[7] J.-D. Boissonnat, O. Devillers, M. Teillaud, and M. Yvinec, Triangulations in CGAL, *Proc. 16th annual ACM symposium on computational geometry* (2000), 11–18.

[8] Boost, Website: `http://www.boost.org/` (2008).

[9] H. Brönnimann, C. Burnikel, and S. Pion, Interval arithmetic yields efficient dynamic filters for computational geometry, *SCG '98: Proceedings of the fourteenth annual symposium on computational geometry*, ACM (1998), 165–174.

[10] H. Brönnimann and M. Yvinec, Efficient exact evaluation of signs of determinants, *Algorithmica* **27**, 1 (2000), 21–56.

[11] CGAL, Website: `http://www.cgal.org/` (2008).

[12] A. K. Cline and R. J. Renka, A storage-efficient method for construction of a Thiessen triangulation, Technical report (1982).

[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*, Second Edition, MIT Press (2003).

[14] O. Devillers, Improved incremental randomized Delaunay triangulation, *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, ACM (1998), 106–115.

[15] P.-O. Fjällström, J. Katajainen, and J. Petersson, Algorithms for the all-nearest-neighbors problem, Technical report, Department of Computing, University of Copenhagen (1992).

[16] S. Fortune, A sweepline algorithm for Voronoi diagrams, *SCG '86: Proceedings of the second annual symposium on computational geometry*, ACM (1986), 313–322.

[17] S. Fortune, Introduction, *Algorithmica* **27**, 1 (2000), 1–4.

[18] J. H. Friedman, J. L. Bentley, and R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, *ACM Transactions on mathematical software* **3**, 3 (1977), 209–226.

[19] ENS Lyon, Website: `http://lipforge.ens-lyon.fr/www/gappa/` (2008).

[20] GNU MP, Website: `http://gmplib.org/` (2008).

[21] D. Goldberg, What every computer scientist should know about floating-point arithmetic, *ACM Computing surveys* **23**, 1 (1991), 5–48.

[22] J. E. Goodman and J. O'Rourke (Editors), *Handbook of discrete and computational geometry*, Second Edition, CRC Press (2004).

[23] B. J. Gough, An introduction to gcc. Floating-point issues, Webdokument (N/A) `http://www.network-theory.co.uk/docs/gccintro/gccintro_70.html`.

[24] P. J. Green and R. Sibson, Computing Dirichlet tessellations in the plane, *The Computer Journal* **21**, 2 (1978), 168–173.

[25] L. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Transactions on Graphics* **4**, 2 (1985), 74–123.

[26] L. J. Guibas, D. E. Knuth, and M. Sharir, Randomized incremental construction of Delaunay and Voronoi diagrams, *Algorithmica* **7**, 4 (1992), 381–413.

[27] M. T. Heath, *Scientific computing: An introductory survey*, McGraw-Hill Higher Education (2002).

[28] K. Hinrichs, J. Nievergelt, and P. Schorn, A sweep algorithm and its implementation: The all-nearest-neighbors problem revisited, *Proceedings of the 14th international workshop on graph-theoretic concepts in computer science*, Springer-Verlag (1989), 442–457.

[29] Ø. Hjelle and M. Dæhlen, *Triangulations and applications*, Springer (2006).

[30] C. M. Hoffmann, The problems of accuracy and robustness in geometric computation, *Computer* **22**, 3 (1989), 31–40.

[31] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick, Towards implementing robust geometric computations, *Proceedings of the 4th annual symposium on computational geometry*, ACM (1988), 106–117.

[32] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap, Classroom examples of robustness problems in geometric computations, *Computational geometry: Theory and applications* **40**, 1 (2008), 61–78.

[33] D. E. Knuth, *The art of computer programming, vol. 2*, Third Edition, Addison-Wesley (1998).

[34] K. Mehlhorn and S. Näher, *LEDA: A platform for combinatorial and geometric computing*, Cambridge University Press (1999).

[35] K. Mehlhorn and C. Yap, Robust geometric computation, Webdokument (2004) `http://www.cs.nyu.edu/yap/bks/egc/`.

[36] G. Melquiond and S. Pion, Formal certification of arithmetic filters for geometric predicates, *Proc. 17th IMACS World Congress on Scientific , Applied Mathematics and Simulation* (2005).

[37] E. P. Mücke, I. Saias, and B. Zhu, Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations, *SCG '96: Proceedings of the twelfth annual symposium on computational geometry*, ACM (1996), 274–283.

[38] F. P. Preparata and M. I. Shamos, *Computational geometry: An introduction*, Springer-Verlag (1985).

[39] The Geometry Center, Minneapolis, Website: `http://www.qhull.org` (2003).

[40] S. Schirra, Robustness and precision issues in geometric computation, *Handbook on computational geometry*, Elsevier (2000), 597–632.

[41] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, *Discrete and computational geometry* **18**, 3 (1997), 305–363.

[42] J. R. Shewchuk, Triangulation software, Website: `http://www.cs.cmu.edu/~quake/triangle.html` (2005).

[43] J. R. Shewchuk, Lecture notes on geometric robustness, Webdokument (2006) `http://www.cs.berkeley.edu/~jrs/meshpapers/robnotes.ps.gz`.

[44] R. F. Sproull, Refinements to nearest-neighbor searching in k-dimensional trees, *Algorithmica* **6**, 4 (1991), 579–589.

[45] B. Stroustrup, *The C++ programming language*, Addison-Wesley (2000).

[46] K. Sugihara and M. Iri, Construction of the Voronoi diagram for 'one million' generators in single-precision arithmetic, *Proceedings of the IEEE* **80**, 9 (1992), 1471–1484.

[47] K. Sugihara and M. Iri, Topology-oriented implementation — an approach to robust geometric algorithms, *Algorithmica* **27**, 1 (2000), 5–20.

[48] D. Vandevoorde and N. M. Josuttis, *C++ templates: The complete guide*, Addison-Wesley (2006).

[49] C. K. Yap, Robust geometric computation, *Handbook of discrete and computational geometry*, Second Edition, CRC Press (2004), 927–952.