Indoor positioning met Arduino's

Bram Leenders & Patrick van Looy

18 juli 2014

1 Inleiding

Voor veel toepassingen van draadloze sensornetwerken is het weten van de locatie van de sensoren erg nuttig of zelfs noodzakelijk. Een voorbeeld hiervan is een sensornetwerk in het bos, bedoeld om bosbranden te detecteren. Voor geografisch grote netwerken is het een handige toevoeging als een node niet alleen detecteert dat er brand is, maar ook waar deze brand is. Op deze manier kan de brand doelgericht en snel bestreden worden, omdat er meer informatie over bekend is.

Locatiebepaling kan op verschillende manieren gedaan worden, een bekend voorbeeld hiervan is GPS. Dit onderzoek legt de focus op locatiebepaling door middel van hoogfrequent geluid. Het doel van dit onderzoek is om de locatie van een Arduino met een ultrasoon-ontvanger te bepalen ten opzichte van vier ultrasoon-verzenders die zich op bekende posities bevinden en afwisselend een geluidspuls versturen. Hiervoor moet een lokalisatiealgoritmeontwikkeld worden dat gebruik maakt van de signalen die verzonden worden door deze vier bakens.

Allereerst zullen we een probleemstelling formuleren zodat we een uitgangspunt voor onze tests hebben, dit doen we in sectie 2. In sectie 3 worden drie mogelijke manieren van afstandbepaling toegelicht. Daarnaast wordt in sectie 4 de implementatie beschreven. Vervolgens behandelen we in sectie 5 de resultaten die we door middel van onze tests hebben gekregen. Als laatste trekken we hieruit een conclusie in sectie 6.

2 Probleemstelling

Het doel van dit onderzoek is het ontwikkelen van een systeem dat positiebepaling op korte afstand kan doen. Met korte afstand wordt een afstand in de orde van 20 meter tot de zenders bedoeld. Hierbij kan gedacht worden aan positiebepaling in een huis of (fabrieks) hal, maar niet aan GPS-achterige applicaties waar wereldwijde of landelijke dekking vereist is.

In 1 zin zeggen wat we doen: Kunnen we met de ons gegeven hardware een positioneringssysteem maken dat op 20 cm nauwkeurig is?

3 Gerelateerd werk

Toelichten waarom we TOF gekozen hebben.

Er zijn verschillende manier om met behulp van radio en/of geluidssignalen een afstand te meten, we zullen de volgende drie kort toelichten:

- Received Signal Strength Indication (RSSI)
- Time Difference of Arrival (TDOA)
- Time of Flight (TOF)

Deze drie zijn met de beschikbare hardware (Arduino, RF24 chip en microfoon) implementeerbaar, dus er moet een keuze uit deze drie gemaakt worden.

3.1 Received Signal Strength Indication

Bij RSSI wordt de sterkte van het signaal gebruikt om een schatting te maken van de afstand tussen een zender en een ontvanger. Deze methode heeft een aantal nadelen, zoals beschreven door Seshadi et al. [4]. De belangrijkste nadelen zijn de wisselende signaalsterkte, kosten van meet- en zendapparatuur en verstoringen van objecten tussen zender en ontvanger. Vanwege deze redenen hebben we niet voor RSSI als methode gekozen.

3.2 Time Difference of Arrival

Bij TDOA wordt gebruik gemaakt van het verschil in afstand tussen twee zenders. Als twee zenders tegelijkertijd een signaal uitzenden, kan een ontvanger een mogelijk verschil in ontvangsttijd meten. Dit verschil in ontvangsttijd kan dan omgezet worden naar een verschil in afstand tussen de twee zenders. Deze methode wordt verder uitgewerkt door Gustaffson et al. [2].

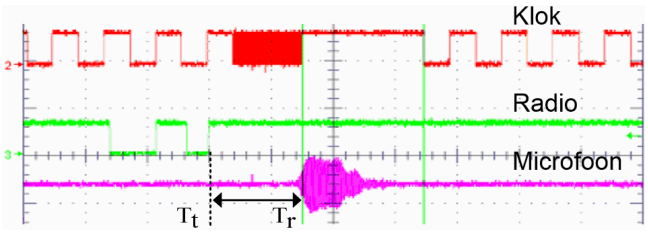
In dit onderzoek kan TDOA niet worden gebruikt, maar hier is niet voor gekozen omdat het bijhouden van het verschil tussen ontvangsttijden op de ontvanger (in dit geval een Arduino) niet erg nauwkeurig is. Om dit algoritme goed uitvoerbaar te laten zijn dienen eigenlijk de beacons als ontvangers gebruikt worden, en de te positioneren Arduino als zender. Echter, gegeven de opstelling die in dit onderzoek gebruikt wordt kunnen de beacons alleen als zenders gebruikt worden.

3.3 Time of Flight

Voor het onderzoek van deze paper is de time of flight (TOF) gebruikt om de afstand tussen beacons en de ontvanger te meten. Deze methode maakt ook gebruik van het verschil in ontvangsttijd van twee signalen. Echter, niet tussen signalen van twee nodes, maar tussen twee types signalen: radio en geluidssignalen.

TOF maakt gebruik van het verschil in propagatiesnelheid van licht en geluid; radiosignalen gaan met lichtsnelheid (ca. $3 \cdot 10^8$m/s), maar geluid gaat veel langzamer (ca 340 m/s). Door beacons tegelijkertijd een radio- en een geluidssignaal uit te laten zenden kan met behulp van het verschil in ontvangsttijden de afstand tussen het beacon en een ontvanger berekend worden. Deze techniek wordt beschreven door Barshan en Ballur [1].

Figuur 1 geeft een voorbeeld van inkomende signalen bij een dergelijke aanpak.



Figuur 1: Tijdsdiagram radio en microfoon input. *Bron: [3]*

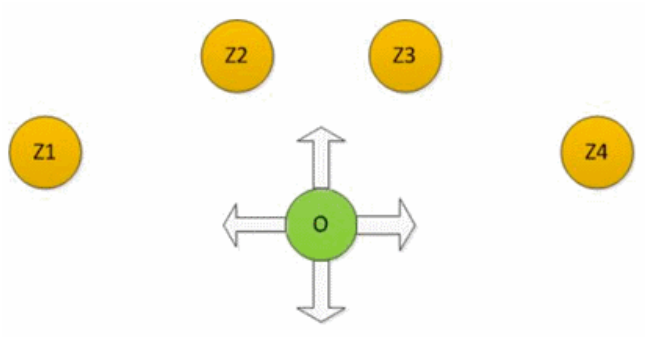
4 Implementatie

De implementatie gebruikt de time of flight (TOF) om de afstand tussen beacons en de ontvanger te meten.

4.1 Opstelling voor positiebepaling met hoogfrequent geluid

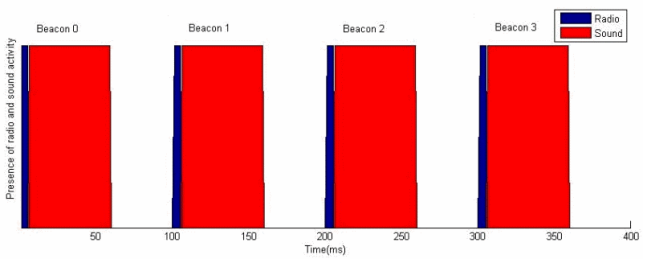
Aparte section: testopstelling

De opstelling voor het bepalen van de positie bestaat uit vier bakens, die zijn genummerd van 0 tot 3. Elk baken bestaat uit een Arduino mini met daarop aangesloten een NRF2401L+-radio en een ultrasoon-zender. De bakens zijn bevestigd op een statief. De locaties van de bakens is bekend. Een voorbeeld van een opstelling met de vier bakens en een ontvanger is te zien in afbeelding 2.



Figuur 2: Opstelling met vier zenders en een ontvanger.

De activiteiten van de verschillende bakens zijn als volgt: een van de bakens (Baken 0) verstuurt radioberichten met een interval van 100 ms. Dit bericht bestaat uit een uint8 (een getal tussen 0 en 255) waarin het nummer staat van de baken die aan de beurt is voor het versturen van een geluidspuls. (Baken 0 stuurt ook een bericht als baken 0 zelf aan de beurt is.) Als een baken een bericht ontvangt waarin zijn identicatienummer staat, verstuurt deze direct hierna een geluidspuls op een frequentie van circa 40kHz en met een duur van 50 ms. De bakens zijn dus nooit tegelijkertijd actief. Figuur 3 toont de sequentie van activiteiten van de verschillende bakens. Verder zijn in tabel 1 de verschillende instellingen te zien waarop de radios onderling communiceren.



Figuur 3: Opstelling met vier zenders en een ontvanger.

Kanaal	76 (standaard RF24 instelling)
Automatisch herverzenden	Uit
Transmissiesnelheid	2 Mbps
Adres verzendende pipe	0xdeadbeefa1LL
Payload-grootte	1 byte

Tabel 1: Instellingen radio

4.2 Algoritme

Voor het omzetten van de afstanden tussen de ontvanger en de verschillende beacons naar een positie zijn de berekeningen van Park et al. [3] gebruikt.

Berekeningen kort uitleggen: wat doen we precies? (matrices laten zien)

Doordat de afstanden tot de beacons bekend zijn kan een cirkel worden afgeleid waarop de ontvanger zich bevindt. Doordat meerdere cirkels bekend zijn kan de intersectie van die verschillende cirkels berekend worden: deze intersectie is de locatie van de ontvanger.

Er is gekozen voor deze berekening omdat de locaties van de beacons en de afstanden tot de beacons vrij intuitief als matrices gerepresenteerd worden. Als kanttekening moet vermeld worden dat dit algoritme niet alleen de x en y positie berekent, maar ook de z positie probeert te berekenen. Echter, omdat alle beacons op dezelfde hoogte staan is dit met onze opstelling niet mogelijk geweest. Er is immers niet te achterhalen of de z positie positief of negatief is; doordat alle beacons in een vlak staan kan de z-coördinaat gespiegeld worden. Vanwege deze reden hebben we ervoor gekozen om de z-coördinaat niet in de resultaten op te nemen.

De gebruikte code is bijgevoegd in appendix A. Merk op dat de posities van de beacons hardcoded is; deze kan worden aangepast indien de beacons een andere configuratie hebben. De positie van de ontvanger is relatief ten opzichte van de beacons.

Tijdens de tests is de lijn tussen beacon 1 en 2 als y-as gebruikt, en beacon 0 was op de x-as (die onder een hoek van 90 graden de y-as kruist). Het algoritme ondersteunt ook negatieve waardes voor als de ontvanger zich achter de y-as bevindt.

4.3 Toelichting software implementatie

In deze sectie wordt toegelicht hoe het abstracte algotime uit sectie 3.3 en 4.2 geïmplementeerd is.

Zoals gezegd in sectie 3.3 wordt gebruik gemaakt van het verschil in propagatiesnelheid van radiogolven (lichtsnelheid) en geluid (ca 340 m/s) om de afstand tot een baken te bepalen. In onze implementatie wordt eerst gewacht tot er een radiosignaal ontvangen is en de ontvangsttijd wordt genoteerd. Vervolgens wordt gewacht tot het bijbehorende (en tegelijkertijd verzonden) geluidssignaal ontvangen is, en de tijd daarvan wordt genoteerd.

Het systeem heeft een vaste grenswaarde om te bepalen of er een geluidssignaal ontvangen wordt: alle input die die waarde overschrijdt, wordt als signaal gezien. De grenswaarde is circa tien maal de maximale waarde van het ontvangen ruis, en hardcoded in het programma gezet. Deze aanpak bleek een betere nauwkeurigheid te geven dan een gemiddelde input sterkte bijhouden en de afwijking ten opzichte daarvan te bepalen, omdat er vrijwel geen ruis is bij 40 kHz en de extra berekeningen benodigd voor het berekenen van een gemiddelde leiden tot een minder nauwkeurige tijdswaarneming.

Het algoritme van Park et al. [3] gebruikt slechts drie bakens om een positie te bepalen. Onze opstelling heeft vier bakens, wat mogelijkheid voor extra nauwkeurigheid biedt. Het systeem voert het algoritme van Park et al. vier maal uit; eenmaal zonder baken 0, eenmaal zonder baken 1, etcetera. Dit levert vier mogelijke locaties op, waarvan vervolgens het gemiddelde wordt berekend. Het idee achter deze aanpak is dat fouten in de metingen elkaar zo uitmiddelen en tot een nauwkeuriger resultaat leiden.

4.3.1 Glijdend gemiddelde van metingen

Om ook meerdere meetrondes elkaars nauwkeurigheid te laten verbeteren, ondersteunt onze implementatie ook de mogelijkheid om een soort glijdend gemiddelde voor de locatie bij te houden. Hierbij heeft de meest recente meting het zwaarste gewicht (e.g. p, met $0 \leq p \leq 1$), en alle oude metingen samen $1 - p$. Een meting die n meetronden geleden heeft plaatsgevonden, is het gewicht p^n.

Deze methode is efficient in het geheugengebruik, omdat er slechts één variabele bijgehouden hoeft te worden. Wanneer gebruik wordt gemaakt van een glijdend gemiddelde waarbij het gemiddelde van de laatste m metingen gebruikt wordt, moeten m waardes bijgehouden worden. Voor grote waardes van m kan dit problemen opleveren, zeker omdat het gekozen platform (Arduino Uno) slechts een zeer beperkte hoeveelheid RAM heeft.

Om te voorkomen dat één verkeerde meting (bijvoorbeeld $(1000, 1000)$ waarbij de echte locatie $(5, 5)$ is) lang effect heeft, is gekozen voor een methode waarbij metingen die meer dan 25% afwijken van het gemiddelde worden afgezwakt tot de 25% afwijking.

Het nadeel van deze methode is dat bij snel bewegende objecten de locatiebepaling achterloopt op de echte locatie. In dit geval kan de waarde van p hoger gekozen worden: bij een waarde van $p = 1$ wordt alleen nog de laatste meting gebruikt.

5 Resultaten en discussie

Tijdens het onderzoek is een belangrijke beperking van de Arduino als platform: de beperkte hoeveelheid RAM. Wanneer er meer dan twee matrices in het algoritme gebruikt worden, kan dit tot een out of memory error leiden. Deze wordt niet gedetecteerd door de Arduino maar leidt tot het verwijzen naar verkeerde geheugenadressen. Hierdoor worden waardes in het geheugen overgeschreven waardoor de uiteindelijke output van het algoritme niet klopt.

Mochten er meerdere matrices nodig zijn voor het berekeken van een positie, dan is dit wel redelijk gemakkelijk op te lossen. Dit kan namelijk door dezelfde matrix te vervangen nadat de benodigde waarde al berekend is. Zo doet het algoritme wat gebruikt is bij dit onderzoek het ook.

De uiteindelijke metingen, waarbij slechts twee matrices in het algoritme gebruikt werden, waren zeer accuraat. De gemeten positie verschilde in een gecontroleerde testruimte niet meer dan vijf centimeter ten opzichte van de daadwerkelijke positie.

Meer uitleg; drie meetpunten op circa tien cm nauwkeurig, één grote afwijker, een heel nauwkeurig (5cm) in gecontroleerde omgeving

Hierbij moet worden opgemerkt dat de opstelling geoptimaliseerd was voor deze ruimte met constante temperatuur, weinig ruis en geen wind.

Bij het implementeren van het algoritme is nog geen rekening gehouden met het stroomverbruik van de Arduino. Voor veel draadloze sensornetwerken is de hoeveelheid beschikbare energie vaak beperkt en kan dit wel een belangrijk aspect van een algoritme zijn. Een volgend onderzoek zou verder kunnen uitlichten hoe hiermee rekening mee kan worden gehouden.

6 Conclusie

We hebben laten zien dat met eenvoudige hardware een accurate positiebepaling gedaan kan worden op de korte afstand. Hierbij is gebruik gemaakt van het Time of Flight principe, wat in een gebied van tientallen vierkante meters een afwijking van minder dan 5 cm heeft.

Voor implementaties waarbij dit stroomverbruik geen rol speelt kan dit algoritme dus gebruikt worden om nauwkeurige metingen te doen. Als stroomverbruik wel belangrijk is, is het mogelijk om bijvoorbeeld minder meetrondes te doen; bijvoorbeeld niet constant, maar om de seconde.

Stroomverbruik is natuurlijk belangrijk wanneer we in het veld werken!

Referenties

[1] Billur Barshan. Fast processing techniques for accurate ultrasonic range measurements. *Measurement Science and technology*, 11(1):45, 2000.

[2] Fredrik Gustafsson and Fredrik Gunnarsson. Positioning using time-difference of arrival measurements. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, volume 6, pages VI–553. IEEE, 2003.

[3] Jaehyun Park, Sunghee Choi, and Jangmyung Lee. Beacon scheduling algorithm for localization of a mobile robot. In *Intelligent Robotics and Applications*, pages 594–603. Springer, 2011.

[4] Vinay Seshadri, Gergely V Zaruba, and Manfred Huber. A bayesian sampling approach to in-door localization of wireless devices using received signal strength indication. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pages 75–84. IEEE, 2005.

A Arduino code

```
/*
    Positioning system for Arduino One with RF24 radio chip
*/
#include <SPI.h>
#include "nRF24L01.h"
#include "RF24.h"
#include "printf.h"
#include "MatrixMath.h"

#define N (3)
// Kunstmatige waarde voor Z coordinaten
#define Z 1.0
// Percentage verschil (0 < MAX_DIFF <= 1) dat tussen twee metingen mag zitten.
#define MAX_DIFF (0.25)
// Percentage dat de nieuwste meting in het gemiddelde meetelt (0 < WEIGHT <= 1)
// Bij WEIGHT=1 wordt er geen gemiddelde bijgehouden, maar is de nieuwste meting de enige die meetelt.
#define WEIGHT (0.2)

RF24 radio(3, 9);
unsigned long radiotime;
unsigned long audiotime;
unsigned long timelimit = 50000LL;
uint8_t activeBeacon;

float pos[4][2] = { // Positions van de beacons; pos[1][1] is de y positie van beacon 1
    {0.0, 75.0},
    {72.0, 0.0},
    {294.0, 0.0},
    {372.0, 136.0}
};

float D[4];

void setup() {
  // initialize the serial communication:
  Serial.begin(9600);
  printf_begin();

  // Setup and configure rf radio
  radio.begin();
  radio.setRetries(0,0);

  radio.setDataRate(RF24_2MBPS);
  radio.setChannel(76);
  radio.setPayloadSize(1);
  radio.openReadingPipe(1, 0xdeadbeefa1LL);
  radio.openWritingPipe(0xdeadbeefa1LL);
  radio.startListening();
  radio.setAutoAck(false);
}

void loop() {
  while(radio.available()) {
    radio.read(&activeBeacon, sizeof(uint8_t));
```

```
55    }
56    while (! radio.available());
57
58    radiotime = micros();
59    radio.read( &activeBeacon, sizeof(uint8_t));
60
61    if(activeBeacon > 3) { return; }
62
63    while(analogRead(A0) < 50) {
64      audiotime = micros();
65      if(audiotime - radiotime > timelimit) {
66        return;
67      }
68    }
69
70    float diff = audiotime - radiotime;
71    diff = diff * 0.03432; // Afstand tot beacon in cm
72
73    //Zwak uitschieters een beetje af: max 30% increase
74    if(diff > (D[activeBeacon]* (1.0 + MAX_DIFF)) && D[activeBeacon↩
          ] > 0) {
75      diff = D[activeBeacon] * (1.0 + MAX_DIFF);
76    }
77
78    if(diff < (D[activeBeacon]* (1.0 - MAX_DIFF))) {
79      diff = D[activeBeacon]*(1.0 - MAX_DIFF);
80    }
81
82    D[activeBeacon] = D[activeBeacon]*(1.0 - WEIGHT) + diff*WEIGHT;↩
          // Weer schuivend gemiddelde */
83
84    if(activeBeacon == 3) {
85      calcPosition();
86    }
87  }
88
89  float A[N][N];
90  float B[N];
91
92  void calcPosition() {
93      // Relatieve afstanden tussen de nodes; gebruikt node 3 nog ↩
            niet!
94      A[0][0] = 2*pos[1][0] - 2*pos[0][0]; A[0][1] = 2*pos[1][1] - ↩
            2*pos[0][1]; A[0][2] = Z;
95      A[1][0] = 2*pos[2][0] - 2*pos[1][0]; A[1][1] = 2*pos[2][1] - ↩
            2*pos[1][1]; A[1][2] = Z;
96      A[2][0] = 2*pos[0][0] - 2*pos[2][0]; A[2][1] = 2*pos[0][1] - ↩
            2*pos[2][1]; A[2][2] = Z;
97      Matrix.Invert((float*)A,N);
98      B[0] = (D[0]*D[0]) - (D[1]*D[1]) - (pos[0][0]*pos[0][0]) + (↩
            pos[1][0]*pos[1][0]) - (pos[0][1]*pos[0][1]) + (pos↩
            [1][1]*pos[1][1]);
99      B[1] = (D[1]*D[1]) - (D[2]*D[2]) - (pos[1][0]*pos[1][0]) + (↩
            pos[2][0]*pos[2][0]) - (pos[1][1]*pos[1][1]) + (pos↩
            [2][1]*pos[2][1]);
100     B[2] = (D[2]*D[2]) - (D[0]*D[0]) - (pos[2][0]*pos[2][0]) + (↩
            pos[0][0]*pos[0][0]) - (pos[2][1]*pos[2][1]) + (pos↩
            [0][1]*pos[0][1]);
101     float P3[N];
102     Matrix.Multiply((float*)A,(float*)B,N,N,1,(float*)P3);
103
```

```
104     // Relatieve afstanden tussen de nodes; gebruikt node 2 nog ↩
            niet!
105     A[0][0] = 2*pos[1][0] - 2*pos[0][0]; A[0][1] = 2*pos[1][1] - ↩
            2*pos[0][1]; A[0][2] = Z;
106     A[1][0] = 2*pos[3][0] - 2*pos[1][0]; A[1][1] = 2*pos[3][1] - ↩
            2*pos[1][1]; A[1][2] = Z;
107     A[2][0] = 2*pos[0][0] - 2*pos[3][0]; A[2][1] = 2*pos[0][1] - ↩
            2*pos[3][1]; A[2][2] = Z;
108     Matrix.Invert((float*)A,N);
109     B[0] = (D[0]*D[0]) - (D[1]*D[1]) - (pos[0][0]*pos[0][0]) + (↩
            pos[1][0]*pos[1][0]) - (pos[0][1]*pos[0][1]) + (pos↩
            [1][1]*pos[1][1]);
110     B[1] = (D[1]*D[1]) - (D[3]*D[3]) - (pos[1][0]*pos[1][0]) + (↩
            pos[3][0]*pos[3][0]) - (pos[1][1]*pos[1][1]) + (pos↩
            [3][1]*pos[3][1]);
111     B[2] = (D[3]*D[3]) - (D[0]*D[0]) - (pos[3][0]*pos[3][0]) + (↩
            pos[0][0]*pos[0][0]) - (pos[3][1]*pos[3][1]) + (pos↩
            [0][1]*pos[0][1]);
112     float P2[N];
113     Matrix.Multiply((float*)A,(float*)B,N,N,1,(float*)P2);
114
115     // Relatieve afstanden tussen de nodes; gebruikt node 1 nog ↩
            niet!
116     A[0][0] = 2*pos[2][0] - 2*pos[0][0]; A[0][1] = 2*pos[2][1] - ↩
            2*pos[0][1]; A[0][2] = Z;
117     A[1][0] = 2*pos[3][0] - 2*pos[2][0]; A[1][1] = 2*pos[3][1] - ↩
            2*pos[2][1]; A[1][2] = Z;
118     A[2][0] = 2*pos[0][0] - 2*pos[3][0]; A[2][1] = 2*pos[0][1] - ↩
            2*pos[3][1]; A[2][2] = Z;
119     Matrix.Invert((float*)A,N);
120     B[0] = (D[0]*D[0]) - (D[2]*D[2]) - (pos[0][0]*pos[0][0]) + (↩
            pos[2][0]*pos[2][0]) - (pos[0][1]*pos[0][1]) + (pos↩
            [2][1]*pos[2][1]);
121     B[1] = (D[2]*D[2]) - (D[3]*D[3]) - (pos[2][0]*pos[2][0]) + (↩
            pos[3][0]*pos[3][0]) - (pos[2][1]*pos[2][1]) + (pos↩
            [3][1]*pos[3][1]);
122     B[2] = (D[3]*D[3]) - (D[0]*D[0]) - (pos[3][0]*pos[3][0]) + (↩
            pos[0][0]*pos[0][0]) - (pos[3][1]*pos[3][1]) + (pos↩
            [0][1]*pos[0][1]);
123     float P1[N];
124     Matrix.Multiply((float*)A,(float*)B,N,N,1,(float*)P1);
125
126     // Relatieve afstanden tussen de nodes; gebruikt node 0 nog ↩
            niet!
127     A[0][0] = 2*pos[2][0] - 2*pos[1][0]; A[0][1] = 2*pos[2][1] - ↩
            2*pos[1][1]; A[0][2] = Z;
128     A[1][0] = 2*pos[3][0] - 2*pos[2][0]; A[1][1] = 2*pos[3][1] - ↩
            2*pos[2][1]; A[1][2] = Z;
129     A[2][0] = 2*pos[1][0] - 2*pos[3][0]; A[2][1] = 2*pos[1][1] - ↩
            2*pos[3][1]; A[2][2] = Z;
130     Matrix.Invert((float*)A,N);
131     B[0] = (D[1]*D[1]) - (D[2]*D[2]) - (pos[1][0]*pos[1][0]) + (↩
            pos[2][0]*pos[2][0]) - (pos[1][1]*pos[1][1]) + (pos↩
            [2][1]*pos[2][1]);
132     B[1] = (D[2]*D[2]) - (D[3]*D[3]) - (pos[2][0]*pos[2][0]) + (↩
            pos[3][0]*pos[3][0]) - (pos[2][1]*pos[2][1]) + (pos↩
            [3][1]*pos[3][1]);
133     B[2] = (D[3]*D[3]) - (D[1]*D[1]) - (pos[3][0]*pos[3][0]) + (↩
            pos[1][0]*pos[1][0]) - (pos[3][1]*pos[3][1]) + (pos↩
            [1][1]*pos[1][1]);
134     float P0[N];
135     Matrix.Multiply((float*)A,(float*)B,N,N,1,(float*)P0);
```

```
136
137     // Bereken het gemiddelde van de verschillende metingen:
138     int avg[N];
139     avg[0] = (int) (P0[0] + P1[0] + P2[0] + P3[0]) / 4.0;
140     avg[1] = (int) (P0[1] + P1[1] + P2[1] + P3[1]) / 4.0;
141     avg[2] = (int) (P0[2] + P1[2] + P2[2] + P3[2]) / 4.0;
142
143     printf("Position: (%d,%d)\n\n", avg[0], avg[1]);
144 }
```