Universitaire Instelling Antwerpen
Departement Wiskunde-Informatica

Stochastisch schatten van de massa van een tandwiel met variërende dichtheid

Sven Maerivoet

2e licentie Informatica
Academiejaar 2000 - 2001

Opdracht voor het vak *Meten en simuleren met computers*
Titularis : Frans Verbeure

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1 Probleemstelling

In dit deel wordt kort en bondig het probleem intuïtief geformuleerd. Ook worden alle parameters die hier een rol bij spelen, toegelicht en gequantifieerd en worden enkele belangrijke veronderstellingen besproken.

1.1 Inleiding

Hansen Transmissions International is een bedrijf dat onder andere tandwielen construeert[1]. Vanuit dit perspectief ontstond de opdracht, namelijk het berekenen van de massa van een tandwiel met behulp van de Monte Carlo Methode. Gekende parameters waren de dimensies van het tandwiel en de massa-dichtheid ρ op verschillende plaatsen in het tandwiel, gevraagd werd om de massa stochastisch te schatten. Als referentie-resultaat wordt daarenboven ook nog eens de analytische oplossing berekend.

1.2 Parameters

Volgende dimensies van het tandwiel zijn gegeven :

- de straal r_1 van het gat (1 cm),
- de straal r_2 van de schijf (5 cm),
- de straal r_3 van de cirkel die de tanden beschrijven (8 cm),
- de dikte d (1 cm),
- de constante massa-dichtheid ρ_{schijf} in de schijf (5 kg/dm^3),
- de variërende massa-dichtheid ρ_{tand} in de tanden (zie vergelijking 1)
- en het aantal tanden K (32).

Volgende uitdrukking wordt gebruikt voor de variërende massa-dichtheid :

$$\rho_{\text{tand}} = 5 \text{ kg/dm}^3 + a \cdot (r - r_2) \tag{1}$$

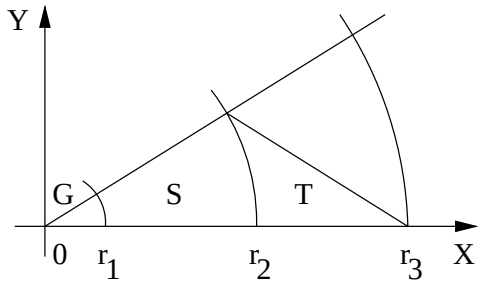
waarbij a een factor gelijk aan 1/2 kg/dm^4 is en r de afstand[2] van de oorsprong tot een punt in een tand (merk op dat er een verschil wordt gebruikt om een continue overgang van de schijf naar een tand te verzorgen).

[1] Voor meer informatie, zie `http://www.hansentransmissions.com`.

[2] Merk op : het betreft hier de afstand geldend in het XY-vlak (dus onafhankelijk van de z-coördinaat).

1.3 Veronderstellingen

Belangrijk is de geometrische configuratie van het tandwiel, zoals te zien op figuur 1 (G is het gat, S is de schijf, T is een (halve) tand en er wordt een rechtshandig assenstelsel gebruikt waarbij de Z-as dus uit het blad komt en naar ons wijst). Het gat G is perfect circulair en de tanden T zijn in essentie langbenige 'driehoeken' die onderaan een cirkelboog vormen en bovenaan eindigen in een punt (de meer realistische tandwielen hebben daarentegen een 'golfvorm').



Figuur 1: De geometrische configuratie van het tandwiel.

Verder is het ook zo dat in punten in de tanden de massa-dichtheid weliswaar lineair varieert met de straal (i.e. de afstand in het XY-vlak tot de oorsprong), maar dat deze massa-dichtheid *constant* is voor punten die *dezelfde* hoek en straal maar een *verschillende* z-coördinaat hebben.

2 Oplossingsmethodes

Als oplossingsmethodes worden twee technieken uitgebreid besproken : de eerste techniek is een analytische berekening van de massa en de tweede techniek steunt op het gebruik van de Monte Carlo Methode voor het stochastisch schatten van de massa.

2.1 Analytisch berekenen

Analytisch berekenen kan eenvoudig gebeuren indien het probleem meetkundig wordt aangepakt en er dan een analytische herformulering van wordt gegeven. Concreet komt het erop neer dat het probleem in deelproblemen wordt opgesplitst :

- de massa van de schijf,
- de massa van een halve tand en
- tenslotte de combinatie van beiden tot de massa van het volledige tandwiel.

2.1.1 De massa van de schijf

De massa van de schijf is zeer eenvoudig te berekenen, het betreft hier immers een cylinder met een gat in. In [LS99b] vinden we dat :

$$V_{\text{cylinder}} = \pi h r^2 \tag{2}$$

waarbij V het volume, h de hoogte en r de straal van de cylinder is.

Berekenen we nu het verschil van het volume V_{schijf} van de schijf S met het volume V_{gat} van het gat G (door toepassing van vergelijking 2 waarin h nu wordt vervangen door de dikte d), dan vinden we :

$$\begin{aligned} V_{\text{schijf}} &= V_{\text{volledige schijf}} - V_{\text{gat}} \\ &= \pi d(r_2^2 - r_1^2) \\ &= 75,398224 \text{ cm}^3. \end{aligned} \tag{3}$$

In de schijf S is de massa-dichtheid constant en daarenboven geldt volgende betrekking (zie [Hew89]) :

$$\rho = \frac{m}{V} \Rightarrow m = \rho \cdot V \tag{4}$$

met ρ de massa-dichtheid, m de massa en V het volume. We kunnen dus de massa van de schijf S als volgt berekenen :

$$\begin{aligned} m_{\text{schijf}} &= \rho_{\text{schijf}} \cdot V_{\text{schijf}} \\ &= 5 \text{ kg/dm}^3 \cdot V_{\text{schijf}} \\ &= 0.005 \text{ kg/cm}^3 \cdot V_{\text{schijf}} \\ &= 0.005 \text{ kg/cm}^3 \cdot 75,398224 \text{ cm}^3 \\ &= 0,3769911 \text{ kg}. \end{aligned} \tag{5}$$

2.1.2 De massa van een tand

Deze analytische berekening heeft wat meer voeten in de aarde en de grootste moeilijkheden zitten in het feit dat de massa-dichtheid nu niet meer constant is en dat de tand niet perfect driehoekig is. De methode die we hier gebruiken, bestaat uit het integreren van massa-elementjes uit de tand T :

$$m = \int_{\text{T}} dm. \tag{6}$$

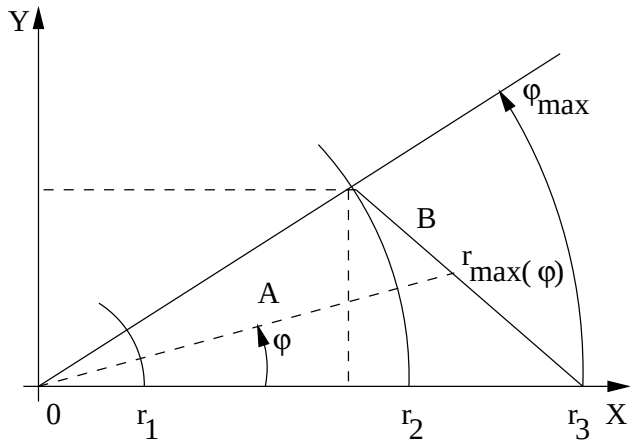
De kunst bestaat er nu in om een geschikte definitie voor het infinitesimaal klein massa-elementje dm te vinden. Om het wat gemakkelijker te maken, schakelen we van carthesische coördinaten over naar cylindrische coördinaten. Bij deze overschakeling mogen we niet vergeten de Jacobiaan van de coördinaat-transformatie mee in rekening te brengen (zie [LS99a]) :

$$\begin{aligned} dm &= \rho_{\text{tand}}(x, y)\, dx\, dy\, dz \\ &= \rho_{\text{tand}}(r, \varphi)\, r\, dr\, d\varphi\, dz. \end{aligned} \tag{7}$$

Het oplossen van vergelijking 6 komt nu neer op het uitwerken van volgende drievoudige integraal (met gepaste grenzen) :

$$m = \int_{z\text{min}}^{z\text{max}} \int_{\varphi\text{min}}^{\varphi\text{max}} \int_{r\text{min}^{(\varphi)}}^{r\text{max}^{(\varphi)}} \rho_{\text{tand}}(r, \varphi)\, r\, dr\, d\varphi\, dz. \tag{8}$$

In vergelijking 8 geldt dat z_{min} gelijk aan 0 en z_{max} gelijk aan 1 is. Er is ook te zien dat de integratiegrenzen $r_{\text{min}}(\varphi)$ en $r_{\text{max}}(\varphi)$ afhankelijk van de hoek φ zijn. Om dit analytisch uit te drukken, werd het snijpunt tussen de rechten A en B uit figuur 2 analytisch uitgedrukt (merk op dat we hier met slechts twee dimensies werken (de straal r en de hoek φ) omdat de massa-dichtheid toch dezelfde is voor punten met dezelfde hoek en straal maar een verschillende z-coördinaat).



Figuur 2: Bepaling van r_{max} in functie van φ.

Stellen we beide vergelijkingen op in carthesische coördinaten, dan krijgen we :

$$\begin{aligned} A &\leftrightarrow \tan(\varphi) \cdot x, && (9)\\ B &\leftrightarrow y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) \\ &\leftrightarrow \frac{r_2 \sin(\varphi_{\text{max}})}{r_2 \cos(\varphi_{\text{max}}) - r_3} \cdot (x - r_3). && (10) \end{aligned}$$

De rechte B gaat door de punten $(r_3, 0)$ en $(r_2 \cos(\varphi_{\text{max}}), r_2 \sin(\varphi_{\text{max}}))$, wat haar vergelijking verklaart. De hoek φ_{max} kan gevonden worden doordat deze een halve tand overspant :

$$\begin{aligned} \varphi_{\text{max}} &= \frac{1}{2} \cdot \frac{2\pi}{K} \\ &= \frac{\pi}{32}. && (11) \end{aligned}$$

Triviaal geldt dat φ_{min} gelijk aan 0 is. Om nu $r_{\text{max}}(\varphi)$ in functie van φ uit te drukken, stellen we de vergelijking van de rechte A gelijk aan die van de rechte B, lossen we deze betrekking op naar x, berekenen dan y en uit x en y berekenen we tenslotte de gezochte $r_{\text{max}}(\varphi)$:

$$\begin{aligned} A &= B && (12)\\ &\Updownarrow \\ \tan(\varphi) \cdot x &= \frac{r_2 \sin(\varphi_{\text{max}})}{r_2 \cos(\varphi_{\text{max}}) - r_3} \cdot (x - r_3) && (13)\\ &\Updownarrow \\ \left(\tan(\varphi) - \frac{r_2 \sin(\varphi_{\text{max}})}{r_2 \cos(\varphi_{\text{max}}) - r_3}\right) \cdot x &= -r_3 \cdot \frac{r_2 \sin(\varphi_{\text{max}})}{r_2 \cos(\varphi_{\text{max}}) - r_3}. && (14) \end{aligned}$$

Om de uitdrukkingen wat leesbaarder te maken, voeren we volgende notatie in :

$$\begin{aligned} \alpha &= r_2 \sin(\varphi_{\text{max}}), && (15)\\ \beta &= r_2 \cos(\varphi_{\text{max}}) - r_3. && (16)\\ & && (17) \end{aligned}$$

We herschrijven vergelijking 14 en vinden dat :

$$\begin{aligned} x &= -\frac{\alpha r_3}{\beta} \cdot \left(\frac{\beta}{\tan(\varphi)\beta - \alpha} \right) \\ &= -\frac{\alpha r_3}{\tan(\varphi)\beta - \alpha}. \end{aligned} \qquad (18)$$

Vermits nu $y = \tan(\varphi) \cdot x$, kunnen we de gezochte uitdrukking voor $r_{\text{max}}(\varphi)$ opstellen :

$$\begin{aligned} r_{\text{max}}(\varphi) &= \sqrt{x^2 + y^2} \\ &= \sqrt{x^2 + (\tan(\varphi) \cdot x)^2} \\ &= x \cdot \sqrt{1 + \tan^2(\varphi)}. \end{aligned} \qquad (19)$$

Vergelijking 8 kan nu opgelost worden, we kennen immers alle parameters. Het met de hand oplossen van deze integraal blijkt echter geen triviale aangelegenheid te zijn, met als gevolg dat besloten werd om *Mathematica 4.0* in te schakelen aangezien deze uitdrukkingen exact formeel (i.e. symbolisch) kan verwerken. Indien alle integratiegrenzen ingevuld worden, dan geeft *Mathematica 4.0* als oplossing (zie paragraaf A.1 in de appendix voor het gebruikte *Mathematica 4.0 notebook*) :

$$m_{\text{halve tand}} = (0,00370238\ldots + (3,48127861\ldots \times 10^{-28})i) \text{ kg}. \qquad (20)$$

Merk op dat dit een complex getal is ! De uitvoer die *Mathematica 4.0* geeft is een gesloten ‘exacte’ uitdrukking, wat maakt dat het resultaat van de exacte berekening nogal ‘gevoelig’ kan zijn. Maar we kunnen altijd de modulus van dit complex getal berekenen (wat een zeer goede benadering is vermits de oplossing toch dicht bij X-as ligt). Hiervoor dienen we de reële en imaginaire componenten te kwadrateren, bij elkaar op te tellen en vervolgens de wortel uit deze som te trekken. De imaginaire component is echter zo klein dat de bijdrage tot het eindresultaat verwaarloosbaar is. Indien we *Mathematica 4.0* de integraal *numeriek* laten oplossen, dan geeft deze uiteindelijk de reële component terug. Voor het resultaat in vergelijking 20 werd *Mathematica 4.0* gevraagd om de integraal *analytisch* op te lossen.

De massa van een volledige tand is dus :

$$\begin{aligned} m_{\text{tand}} &= 2 \cdot 0,00370238\ldots \text{ kg} \\ &= 0,00740476\ldots \text{ kg}. \end{aligned} \tag{21}$$

2.1.3 De massa van het volledige tandwiel

Vermenigvuldigen we vergelijking 21 met het totaal aantal tanden K (= 32) en tellen we er de massa van de schijf (vergelijking 5) bij op, dan verkrijgen we de totale massa van een volledig tandwiel :

$$\begin{aligned} m &= (K \cdot m_{\text{tand}}) + m_{\text{schijf}} \\ &= (32 \cdot 0,00740476 \text{ kg}) + 0,3769911 \text{ kg} \\ &= 0,6139434\ldots \text{ kg}. \end{aligned} \tag{22}$$

2.2 Stochastisch schatten

Het hoofddoel was om de massa van een tandwiel *stochastisch* te schatten. Hierbij wordt gebruik gemaakt van de Monte Carlo Methode die zich baseert op het arbitrair genereren van punten in een gekend volume en vervolgens te tellen hoeveel punten er in het lichaam (tandwiel) zelf vallen.

2.2.1 Algemene werkwijze

De algemene werkwijze is zeer eenvoudig en wordt onder andere geschetst in [Ver01]. Het bepalen van de massa van een lichaam L gebeurt als volgt :

- genereer N punten $p_i(x_i, y_i, z_i)$ in een gekend volume V.
- Bereken voor elk punt p_i de massa-dichtheid-contributie aan de totale massa. Deze contributie ρ_i is 0 als $p_i \in (V \setminus L)$ en ze is (expliciet) te berekenen indien $p_i \in L$.
- De totale massa wordt nu verkregen door alle berekende contributies bij elkaar op te tellen en deze som te vermenigvuldigen met het volume V en te delen door het aantal punten N.

Dus :

$$m = \frac{V}{N} \cdot \sum_{i=1}^{N} \rho_i. \tag{23}$$

2.2.2 De Marsaglia pseudo-random-number generator

Het hart van de Monte Carlo Methode wordt gevormd door het *goed* genereren van willekeurige punten in een gekend volume V. Deze punten worden verkregen door willekeurige waarden te gebruiken voor de x-, y- en z-coördinaten. Het genereren van deze punten gebeurt met behulp van een random-number-generator (*RNG*)[3].

Belangrijk is dat de verdeling van deze willekeurige getallen *voldoende uniform* is en dat de RNG een *grote periode* heeft. Vermits het probleem in C++ geprogrammeerd wordt, kunnen we gebruik maken van de standaardfunctie `int rand()` die een getal tussen 0 en een zekere waarde `RAND_MAX` teruggeeft. Echter, de standaard RNGs die bij compilers meegeleverd worden, zijn meestal – kwalitatief gezien – vrij slecht in het genereren van *goede* random getallen. De laagste-orde bits zijn meestal vrij verdacht (zie [Str97]). Dit laatste komt gemakkelijk aan het licht indien we de RNG telkens één random getal laten genereren en direct daarna de initiële waarde (*seed*) instellen met behulp van de RTC (*real-time clock*) van de computer. Deze RTC genereert elke seconde een andere waarde[4] en de meeste RNGs geven gedurende één seconde constant dezelfde waarde terug. Een succesvolle poging om hieraan te verhelpen wordt gevormd door gebruik te maken van een RNG gebaseerd op het werk van George Marsaglia. De code hiervoor (gebaseerd op [Fog93]) is terug te vinden in de appendix (paragraaf A.4). Het algoritme dat deze RNG gebruikt, is als volgt :

$$\begin{aligned} S &= (2111111111 \cdot X[n-4]) + \\ &\quad (1492 \cdot X[n-3]) + \\ &\quad (1776 \cdot X[n-2]) + \\ &\quad (5115 \cdot X[n-1]) + C, \qquad (24) \\ X[n] &= S \bmod 2^{32}, \qquad (25) \\ C &= \left\lfloor \frac{S}{2^{32}} \right\rfloor . \qquad (26) \end{aligned}$$

Merk op dat de waarde van C in een 80-bits `long double` dient gestockeerd te worden.

2.2.3 De diverse methodes

De algemene werkwijze zoals beschreven in paragraaf 2.2.1 werd gecodeerd in een algemene C++ klasse (`Estimator`) en van deze klasse werd een nieuwe klasse afgeleid (namelijk `Gear-Estimator`).

[3] . . .al zijn ze niet echt random maar eerder pseudo-random, pas met behulp van quantum-computers kunnen we spreken van zuivere random-getallen.

[4] Al kan de RTC wel 18,2 kloktiks per seconde genereren met een fout van 0,055 s (zie [Mae01]).

Alle geïmplementeerde methodes werken op dezelfde manier :

- punten worden willekeurig verdeeld gegenereerd in een zeker volume V.
- De massa-dichtheid in elk punt wordt berekend door te kijken of dit punt in het gat, de schijf, een tand of elders ligt (hiervoor wordt onder andere gebruik gemaakt van de uitdrukking voor $r_{\max}(\varphi)$, terug te vinden in vergelijking 19).

Merk op dat elk experiment een herhaling is van tien schattingen (met behulp van N punten per schatting) waarvan de resultaten worden uitgemiddeld. Ook wordt van elke schatting individueel de tijd gemeten alsmede de totale tijd per experiment (dit dient louter om een idee te krijgen inzake de rekentijd die de computer nodig heeft).

Alle experimenten liepen op een AMD Athlon Thunderbird 900 MHz computer die uitgerust was met 256 Mb geheugen en Red Hat Linux 7.1 draaide. Netwerkverbindingen en andere typische beslommeringen werden uitgeschakeld zodat het volledige potentieel van de machine kon benut worden.

De implementatie van alle methodes kan teruggevonden worden in de appendix (paragraaf A.7).

2.2.4 Methode 1

Het volume V was hier een volle cylinder die in de X- en Y-richting straal r_3 had en in de Z-richting van $-d/2$ tot $+d/2$ liep. Merk op dat er eerst werd gewerkt met een balk die rond het tandwiel zat, wat een omsluitend volume van 256 cm^3 gaf. Het volume van de cylinder paste echter nauwer rond de vorm van het tandwiel en het volume V bedraagt hier (zie [LS99b]) :

$$\begin{aligned} V &= d \cdot \pi r_3^2 \\ &= 201,06193\ldots \text{ cm}^3. \end{aligned} \tag{27}$$

Het correct uniform genereren van punten in dit volume werd gedaan door een punt p_i met willekeurige (weliswaar tot het volume beperkte) x-, y- en z-coördinaten te bepalen, vervolgens te testen of dit punt in de cylinder lag en het tenslotte te behouden of te verwerpen (en een nieuw kandidaat-punt te genereren). Een nadeel van deze methode is wel dat veel rekentijd verloren kan gaan indien het frequent gebeurt dat punten geweigerd en opnieuw berekend moeten worden (dus als veel punten in te groot volume gegenereerd worden).

Elk punt p_i werd uitgedrukt in cylindercoördinaten. Op basis van de straal r kon dan bepaald worden of een punt in het gat ($r < r_1$), in de schijf ($r \geq r_1$ en $r \leq r_2$) of buiten het tandwiel ($r > r_3$) viel.

Indien echter $r > r_2$ en $r \leq r_3$ is, dan wil dit zeggen dat het punt ofwel in een tand ofwel in de ruimte tussen de tanden ligt. Het uitvissen wanneer dit wel en niet het geval is, is een aangename

bezigheid die toch wat voeten in de aarde heeft. De methode die ontwikkeld werd, is robuust en steunt op het feit dat een willekeurige hoek $\varphi \in [0, 2\pi[$ kan gereduceerd worden tot een hoek $\varphi' \in [0, \varphi_{\text{max}}]$. Met behulp van deze laatste hoek φ' kan dan gemakkelijk getest worden of $r \leq r_{\text{max}}(\varphi')$.

Het reduceren van de hoek gebeurt als volgt :

- bepaal in welke tand t het punt ligt.
- Trek van φ vervolgens t keer de hoek af die een tand overspant.
- Bepaal de absolute waarde hiervan opdat de hoek in het eerste kwadrant komt te liggen.

In welke tand t ligt het punt ? Volgende formule wordt gebruikt :

$$t = \left\lfloor \frac{\varphi + \varphi_{\text{max}}}{\frac{2\pi}{K}} \right\rfloor \tag{28}$$

Waarbij de noemer uitdrukt hoeveel radialen één tand overspant. Indien $t = K$ dan wordt t gelijk aan 0, dit maakt dat $t \in \{0, \ldots, K-1\}$.

2.2.5 Methode 2

De tweede methode is grotendeels gebaseerd op de eerste methode, met dat verschil dat er nu een cylinder wordt gebruikt waar een gat met straal r_2 in zit. Dit maakt het volume V kleiner (zie [LS99b]) vermits nu enkel punten worden gegenereerd in de zone waar de tanden zich bevinden :

$$\begin{aligned} V &= (d \cdot \pi) \cdot (r_3^2 - r_2^2) \\ &= 122,52211\ldots \text{ cm}^3. \end{aligned} \tag{29}$$

Voor het bepalen in welk gedeelte van het tandwiel het gegenereerde punt valt, wordt dezelfde werkwijze als in methode 1 gevolgd.

2.2.6 Methode 3

In de derde methode werd gebruik gemaakt van een volle cylinder-sector die een halve tand omvat. Gebaseerd op [LS99b], wordt volgende formule gebruikt :

$$\begin{aligned} V &= \frac{d \cdot \varphi_{\text{max}} \cdot r_3^2}{2} \\ &= \pi \text{ cm}^3. \end{aligned} \tag{30}$$

Merk op dat er nu geen reductie van de hoek φ plaatsvindt aangezien φ al in het interval $[0, \varphi_{\text{max}}]$ ligt.

2.2.7 Methode 4

Deze laatste methode verschilt lichtjes van methode 3 in dat opzicht dat ze nu een kleinere cylinder-sector als volume V gebruikt (er is een 'gat' met straal r_2, net zoals in methode 2). Dit levert een nog kleiner volume V op (zie [LS99b]) :

$$\begin{aligned} V &= \frac{d \cdot \varphi_{\text{max}} \cdot (r_3^2 - r_2^2)}{2} \\ &= 1,91441\ldots \text{ cm}^3. \end{aligned} \tag{31}$$

Net zoals in methode 3 treedt er ook nu geen reductie van de hoek φ op aangezien φ al in het interval $[0, \varphi_{\text{max}}]$ ligt.

3 Resultaten

In dit deel worden de resultaten van de vier verschillende methodes gegeven. Van de 'beste' methode (i.e. die met de kleinste variantie) worden daarenboven nog enkele resultaten gegeven die verband houden met de stochastische aard van de Monte Carlo Methode en de keuze van de grootte van de verzameling gebruikte punten.

3.1 Rekentijden

Elk experiment bestond uit een reeks van tien schattingen en er werden honderd miljoen punten gebruikt per schatting (dus elk experiment gebruikte 1 miljard punten). De rekentijd die de computer per schatting nodig had, varieerde van ongeveer tweehonderd seconden tot ongeveer driehonderd seconden, wat maakt dat de totale rekentijd per experiment ongeveer *dertig tot vijftig minuten* bedroeg.

3.2 Gebruikte statistiek

De statistische formules die gebruikt werden om de gegevens te verwerken, zijn standaardformules en men kan ze onder andere terugvinden in [Rou94]. Hoofdzakelijk worden het *steekproefgemiddelde* en de *empirische variantie* gebruikt.

3.2.1 Steekproefgemiddelde

Als kental van *lokatie*, geldt voor een steekproef $(x_1, \ldots, x_n)$ dat :

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{32}$$

waarbij $\overline{x}$ het (rekenkundig) gemiddelde is.

3.2.2 Empirische variantie

Als kental van *spreiding*, geldt voor een steekproef $(x_1, \ldots, x_n)$ met steekproefgemiddelde $\overline{x}$ (zie paragraaf 3.2.1) dat :

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \tag{33}$$

waarbij s^2 nu de empirische variantie voorstelt. Om de standaardafwijking[5] s te verkrijgen, trekt men gewoon de wortel uit vergelijking 33.

3.3 Resultaten van de vier methodes

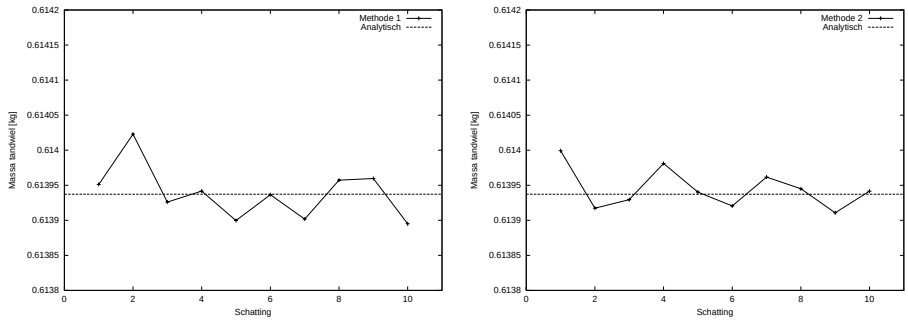
3.3.1 De ruwe data

Alle resultaten werden uitgedrukt in kilogram en de afronding gebeurde tot op acht decimalen na de komma. In tabel 1 is per methode (experiment) te zien wat het resultaat van elk van de tien schattingen was.

Schatting	Methode 1	Methode 2	Methode 3	Methode 4
1	0,61395111	0,61399931	0,61397017	0,61395780
2	0,61402285	0,61391727	0,61400355	0,61393380
3	0,61392601	0,61392951	0,61397137	0,61394713
4	0,61394171	0,61398106	0,61383855	0,61391741
5	0,61389966	0,61394028	0,61407010	0,61395593
6	0,61393650	0,61392051	0,61401396	0,61395619
7	0,61390173	0,61396160	0,61392757	0,61393457
8	0,61395726	0,61394505	0,61386768	0,61393596
9	0,61395963	0,61391073	0,61400423	0,61394706
10	0,61389510	0,61394163	0,61384032	0,61388571

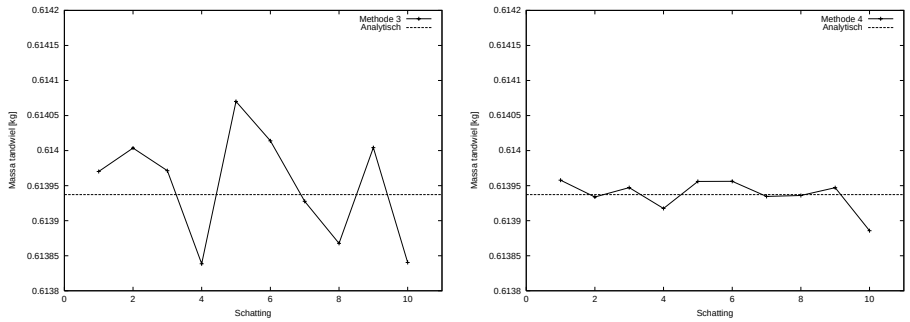
Tabel 1: De resultaten van alle schattingen (kg) van elke methode.

3.3.2 Grafische voorstelling



Figuur 3: Resultaten van methodes 1 en 2 (10^8 punten per schatting).

[5]Merk op dat de standaardafwijking een andere definitie kent dan de standaardfout : bij deze laatste wordt er door n in plaats van door $(n-1)$ gedeeld.



Figuur 4: Resultaten van methodes 3 en 4 (10^8 punten per schatting).

3.3.3 Statistische kentallen

Om een beter idee te krijgen inzake de efficiëntie van elk van de vier methodes, werden het steekproefgemiddelde (zie paragraaf 3.2.1) en de standaardafwijking (zie paragraaf 3.2.2) berekend aan de hand van de in tabel 1 bekomen data. Dit leidde tot het resultaat in tabel 2 :

	Methode 1	Methode 2	Methode 3	Methode 4
Gemiddelde	0,61393916	0,61394470	0,61395075	0,61393716
Standaardafwijking	0,00003798	0,00002856	0,00007952	0,00002210

Tabel 2: De kentallen van alle schattingen (kg) van elke methode.

3.3.4 Conclusies

Kijkend naar tabel 2, merken we dat methode 4 de methode is waarvoor de standaardafwijking het kleinst is, of anders gezegd : voor deze methode is de variantie op de schattingen het kleinst. Een eerste schatting van de massa van het tandwiel levert dus volgend resultaat op :

$$\begin{aligned} m &= \overline{m} \pm s \\ &= (0,61393716 \pm 0,00002210)\ \text{kg}. \end{aligned} \tag{34}$$

Vergelijken we dit met het resultaat dat analytisch werd bekomen (zie vergelijking 22), dan merken we dat :

$$\begin{aligned} |m_{\text{schatting}} - m_{\text{analytisch}}| &= |0,61393716\ \text{kg} - 0,61394340\ \text{kg}| \\ &= 0,00000624\ \text{kg}, \end{aligned} \tag{35}$$

wat al een redelijk nauwkeurige benadering (tot op zes miljoenste) is.

3.4 Invloed van het aantal punten

Uit paragraaf 3.3.4 blijkt dat methode 4 de kleinste variantie heeft. Uitgaande van dit feit, werd onderzocht wat de invloed was van het aantal gegenereerde punten op het eindresultaat bij het gebruik van de Monte Carlo Methode.

3.4.1 Ruwe data

Elk experiment bestond weerom uit een reeks van tien schattingen maar het aantal gebruikte punten varieerde nu per experiment. Er werd telkens met een macht van tien verhoogd, wat aanleiding gaf tot negen experimenten waarbij het *totale* aantal punten per experiment varieerde van tien ($= 10 \cdot 1$) tot tien miljard ($= 10 \cdot 10^9$). De rekentijd ging hierbij *per experiment* van minder dan 1 seconde naar een rekentijd van bijna zeven uur.

In tabellen 3 en 4 is het resultaat van alle schattingen te zien (alle resultaten worden uitgedrukt in kilogram en de afronding gebeurde tot op acht decimalen na de komma).

Schatting	10^0	10^1	10^2	10^3	10^4
1	0,37699110	0,56189639	0,63044683	0,61895400	0,61639308
2	0,37699110	0,56208045	0,63038914	0,60533839	0,61573512
3	0,37699110	0,62539879	0,64951556	0,61453072	0,61628191
4	0,99413756	0,56264849	0,59977369	0,60105724	0,61861782
5	0,37699110	0,62410857	0,56904565	0,62141117	0,61503150
6	0,37699110	0,74923619	0,61189958	0,62636758	0,61320875
7	0,99103288	0,65800900	0,60616402	0,62994633	0,61672896
8	0,37699110	0,56210203	0,61251434	0,60602850	0,61738351
9	0,37699110	0,62439251	0,63690251	0,61720844	0,61673776
10	0,99485582	0,74759584	0,64288870	0,62308843	0,61738336

Tabel 3: De resultaten van alle schattingen (kg) van methode 4 met een verschillend aantal punten.

3.4.2 Statistische kentallen

Voor een beter overzicht werden alle resultaten uitgemiddeld, wat aanleiding gaf tot tabellen 5 en 6.

3.4.3 Grafische voorstelling

Figuren 5, 6, 7, 8 en 9 geven grafisch weer hoe het steekproefgemiddelde zich gedraagt naarmate het aantal gebruikte punten toeneemt (merk op dat er per figuur een andere schaal op de Y-as

Schatting	10^5	10^6	10^7	10^8	10^9
1	0,61635018	0,61390773	0,61400985	0,61395780	0,61393489
2	0,61382615	0,61404486	0,61399862	0,61393380	0,61392960
3	0,61484626	0,61369275	0,61384841	0,61394713	0,61393156
4	0,61280950	0,61407738	0,61386648	0,61391741	0,61394840
5	0,61522745	0,61444206	0,61375753	0,61395593	0,61394262
6	0,61380459	0,61406106	0,61412372	0,61395619	0,61394919
7	0,61434114	0,61364375	0,61382275	0,61393457	0,61393693
8	0,61535275	0,61407090	0,61399246	0,61393596	0,61393546
9	0,61605146	0,61397403	0,61384598	0,61394706	0,61395767
10	0,61310193	0,61473287	0,61384629	0,61388571	0,61393665

Tabel 4: De resultaten van alle schattingen (kg) van methode 4 met een verschillend aantal punten (vervolg).

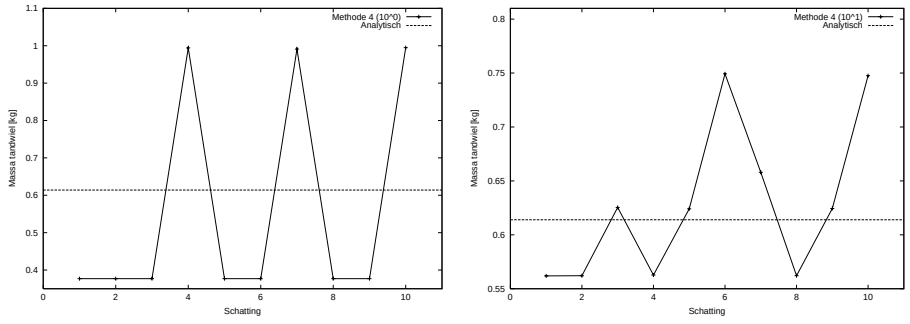
	10^0	10^1	10^2	10^3	10^4
Gemiddelde	0,56189640	0,63044683	0,61895400	0,61639308	0,61635018
Standaardafwijking	0,29772735	0,07420310	0,02404953	0,00959572	0,00147435

Tabel 5: De kentallen van alle schattingen (kg) van methode 4 met een verschillende aantal punten.

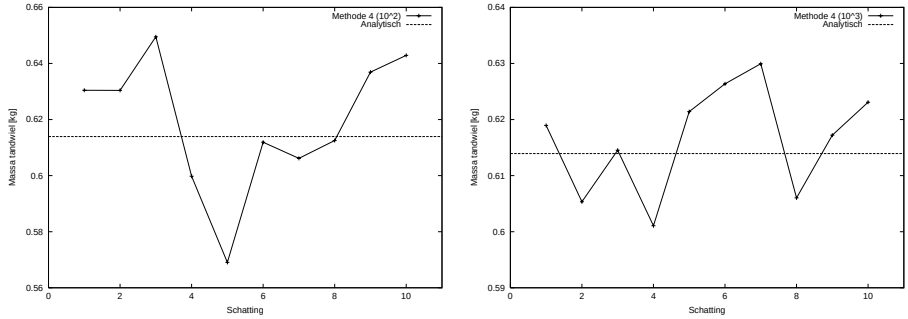
	10^5	10^6	10^7	10^8	10^9
Gemiddelde	0,61457114	0,61406474	0,61391121	0,61393716	0,61394030
Standaardafwijking	0,00119866	0,00032282	0,00011303	0,00002210	0,00000894

Tabel 6: De kentallen van alle schattingen (kg) van methode 4 met een verschillende aantal punten (vervolg).

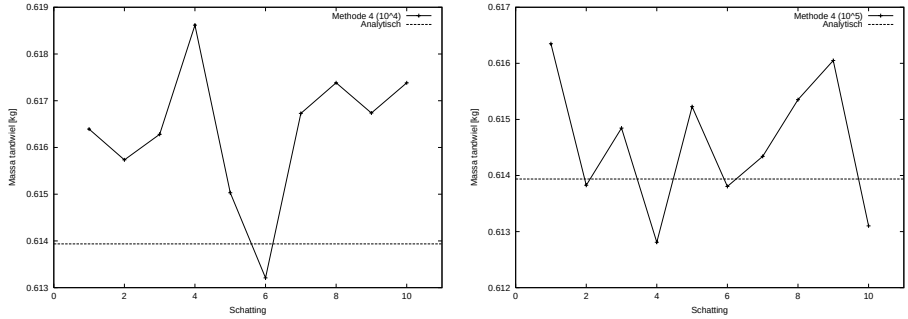
werd genomen, dit om de onderlinge verschillen per schatting gedetailleerder aan het licht te brengen).



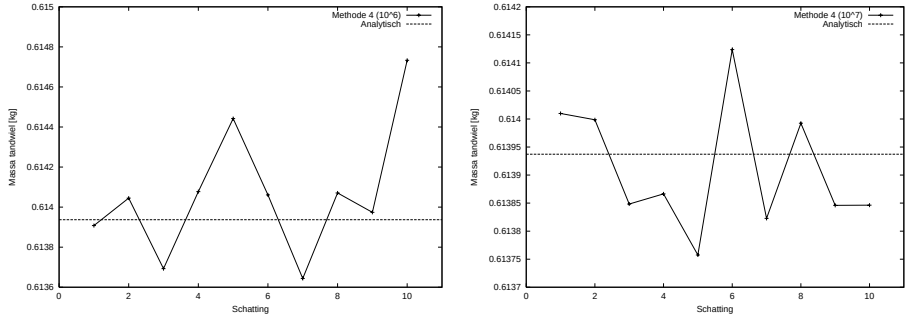
Figuur 5: Resultaten van methode 4 (10^0 en 10^1 punten per schatting).



Figuur 6: Resultaten van methode 4 (10^2 en 10^3 punten per schatting).



Figuur 7: Resultaten van methode 4 (10^4 en 10^5 punten per schatting).

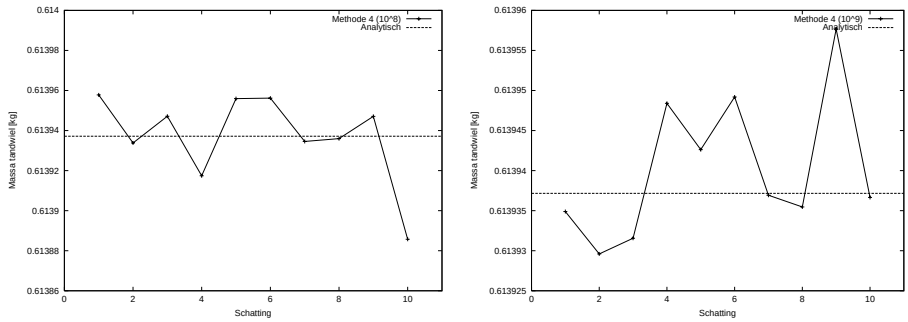


Figuur 8: Resultaten van methode 4 (10^6 en 10^7 punten per schatting).

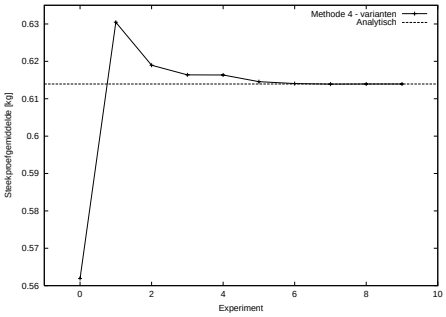
Als we kijken naar figuur 10, dan merken we dat van zodra het aantal punten gebruikt per schatting meer dan 1 miljoen bedraagt, de Monte Carlo Methode al redelijk nauwkeurig de massa van het tandwiel schat. We zien dit nog beter in figuur 11, waar de standaardafwijking voor elk experiment getoond wordt.

3.4.4 Conclusies

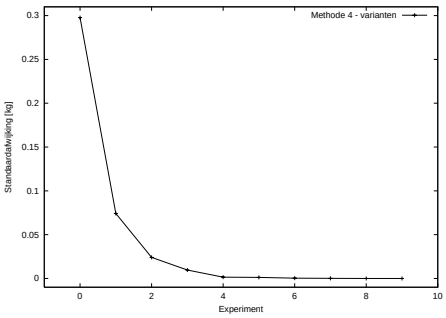
Indien we een totaal van tien miljard punten gebruiken, dan levert de Monte Carlo Methode volgend resultaat op :



Figuur 9: Resultaten van methode 4 (10^8 en 10^9 punten per schatting).



Figuur 10: Resultaten van methode 4 met een variërend aantal punten.



Figuur 11: Standaardafwijkingen van de resultaten van methode 4 met een variërend aantal punten.

$$\begin{aligned} m &= \overline{m} \pm s \\ &= (0,61394030 \pm 0,00000894)\ \text{kg}. \end{aligned} \tag{36}$$

Vergelijken we dit met het resultaat dat analytisch werd bekomen (zie vergelijking 22), dan merken we dat :

$$
\begin{aligned}
|m_{\text{schatting}} - m_{\text{analytisch}}| &= |0,61394030 \text{ kg} - 0,61394340 \text{ kg}| \\
&= 0,00000310 \text{ kg}. \qquad (37)
\end{aligned}
$$

Door tien keer zoveel punten te gebruiken, is de standaardafwijking met ruim een factor 2 geslonken (van 0,00002210 kg naar 0,00000894 kg), maar dit ging wel ten koste van ruim een factor 8 in rekentijd (van 50 minuten naar 7 uur). De uiteindelijke benadering is ook met een factor 2 verbeterd (van 0,00000624 kg naar 0,00000310 kg).

3.5 Conclusies

De conclusie is eenduidig : de Monte Carlo Methode *kan* goede resultaten opleveren (zie paragraaf 3.4.4) die zeer goed aansluiten bij analytisch bekomen waarden. Echter, het verhogen van het aantal gebruikte punten laat de nauwkeurigheid niet drastisch verbeteren, iets dat zeer duidelijk te zien is in figuur 11 : vertrekkende van weinig punten daalt de standaardafwijking zeer snel om dan steeds trager naar 0 te convergeren. Het grote nadeel is dat de rekentijd dan ook *exponentieel* toeneemt. Al kan een oplossing voor dit laatste gezocht worden in meer optimale algoritmes, toch blijft de zojuist vermelde inherente beperking aan de Monte Carlo Methode kleven wat haar algemene bruikbaarheid in het gedrang brengt (zeker indien het real-time toepassingen betreft die nauwkeurige uitvoer moeten leveren).

A Appendix

A.1 Mathematica-notebook

```
Width := 1
Radius1 := 1
Radius2 := 2
Radius3 := 8
NrOfTeeth := 32
ConstantDensity := 5 / 1000
LinearDensityFactor := 0.5 / 10000
MassOfRing := ConstantDensity * Width *
              (Pi * ((Radius2)^2 - (Radius1)^2))

Density[r_] := ConstantDensity +
               (LinearDensityFactor * (r - Radius2))

PhiMin := 0
PhiMax := ((2 * Pi) / NrOfTeeth) * (1 / 2)

Alpha := Radius2 * Sin[PhiMax]
Beta := (Radius2 * Cos[PhiMax]) - Radius3

rMin := Radius2;
rMax[Phi_] := -((Alpha * Radius3) /
               ((Beta * Tan[Phi]) - Alpha)) *
              Sqrt[1 + (Tan[Phi])^2]

RadiusIntegral[Phi_] := Integrate[(Density[r] * r)dr,
                                  {r,rMin,rMax[Phi]}]

AngleIntegral := Integrate[(RadiusIntegral[Phi])dPhi,
                           {Phi,PhiMin,PhiMax}]

WidthIntegral := Integrate[(AngleIntegral)dz,
                           {z,0,Width}]

TotalMass = SetPrecision[MassOfRing +
                         ((2 * NrOfTeeth) * WidthIntegral),53]
```

A.2 Makefile

```
# ------------------------------
# Filename      : Makefile
# Author        : Sven Maerivoet
# Last modified : 03/05/2001
# Target        : make
#
# All rights reserved.
# ------------------------------

CC=g++
LINKFILES=chrono.o estimator.o math.o rng.o

all: estimate-gear-method1 estimate-gear-method2 \
        estimate-gear-method3 estimate-gear-method4

# ---------------
# UTILITY ENTRIES
# ---------------

chrono.o: chrono.h chrono.cpp
$(CC) -c chrono.cpp -o chrono.o

estimator.o: estimator.h estimator.cpp
$(CC) -c estimator.cpp -o estimator.o

math.o: math.h math.cpp
$(CC) -c math.cpp -o math.o

rng.o: rng.h rng.cpp
$(CC) -c rng.cpp -o rng.o

# ---------------
# PROGRAM ENTRIES
# ---------------

estimate-gear-method1: estimate-gear.cpp chrono.o \
        estimator.o math.o rng.o
$(CC) -DMETHOD1 -DNROFSAMPLEPOINTS=100000000 estimate-gear.cpp \
        $(LINKFILES) -o estimate-gear-method1

estimate-gear-method2: estimate-gear.cpp chrono.o \
        estimator.o math.o rng.o
$(CC) -DMETHOD2 -DNROFSAMPLEPOINTS=100000000 estimate-gear.cpp \
        $(LINKFILES) -o estimate-gear-method2

estimate-gear-method3: estimate-gear.cpp chrono.o \
        estimator.o math.o rng.o
$(CC) -DMETHOD3 -DNROFSAMPLEPOINTS=100000000 estimate-gear.cpp \
        $(LINKFILES) -o estimate-gear-method3
```

```
estimate-gear-method4: estimate-gear.cpp chrono.o \
        estimator.o math.o rng.o
$(CC) -DMETHOD4 -DNROFSAMPLEPOINTS=100000000 estimate-gear.cpp \
        $(LINKFILES) -o estimate-gear-method4

# -------------
# CLEAN ENTRIES
# -------------

clean:
rm -f core
rm -f *.o
rm -f calc-modulus
rm -f estimate-gear-method1
rm -f estimate-gear-method2
rm -f estimate-gear-method3
rm -f estimate-gear-method4

clean-results:
rm -f *.txt
```

A.3 Klasse Math

A.3.1 math.h

```
// ------------------------------
// Filename      : math.h
// Author        : Sven Maerivoet
// Last modified : 02/05/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#ifndef SM_MATH_H // include-guard
#define SM_MATH_H

class Math {
  public:
    // useful constants
    static const double kPi;
    static const double kEpsilon;

    // methods
    static double abs(double x);
    static double sqr(double x);
    static double arctan(double y, double x);
};

#endif
```

A.3.2 math.cpp

```
// ------------------------------
// Filename      : math.cpp
// Author        : Sven Maerivoet
// Last modified : 02/05/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#include "math.h"
#include <math.h>

// ----------
// class Math
// ----------

// -----------------------------------------
// initialization of static const class members
```

```
// ---------------------------------------------
const double Math::kPi      = 3.14159265358979323846;
const double Math::kEpsilon = 1E-8;

// -------
// methods
// -------
double Math::abs(double x)
{
  if (x >= 0.0) {
    return x;
  }
  else {
    return -x;
  }
}

double Math::sqr(double x)
{
  return (x * x);
}

double Math::arctan(double y, double x)
{
  double azimuth = kPi / 2.0;

  if (Math::abs(x) > kEpsilon) {
    azimuth = atan(y / x);
    if (x < 0.0) {
      azimuth += kPi;
    }
    else if (y < 0.0) {
      azimuth += (2.0 * kPi);
    }
  }
  else if (y < 0.0) {
    azimuth *= 3.0;
  }

  return azimuth;
}
```

A.4 Klasse RNG

A.4.1 rng.h

```
// ------------------------------
// Filename      : rng.h
// Author        : Sven Maerivoet
// Last modified : 01/05/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#ifndef SM_RNG_H // include-guard
#define SM_RNG_H

typedef unsigned long uint32;

class RNG {
  public:
    // constructor
    RNG(void);

    // destructor
    ~RNG(void);

    // methods
    void setSeed(uint32 seed);
    void setSeedWithTime(void);
    int uniformInt(void);
    double uniform(void);

  private:
    double fX[5];
};

#endif
```

A.4.2 rng.cpp

```
// ------------------------------
// Filename      : rng.cpp
// Author        : Sven Maerivoet
// Last modified : 01/05/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#include "rng.h"
```

```
#include <assert.h>
#include <math.h>
#include <time.h>

/*
  This is a multiply-with-carry type of random number
  generator invented by George Marsaglia.
  The algorithm is :

    S = 2111111111 * X[n-4] +
        1492 * X[n-3] +
        1776 * X[n-2] +
        5115 * X[n-1] +
        C
    X[n] = S modulo 2^32
    C = floor(S / 2^32)

  This implementation uses a long double for C. Note
  that not all C++ compilers support the long double
  (80-bit) floating point format.
*/

// ---------
// class RNG
// ---------

// -----------
// constructor
// -----------
RNG::RNG(void)
{
  // check that the compiler supports 80-bit long doubles
  assert(sizeof(long double) > 9);

  setSeed(0);
}

// ----------
// destructor
// ----------
RNG::~RNG(void)
{
}

// -------
// methods
// -------
void RNG::setSeed(uint32 seed)
{
  // make sure seed != 0
  if (seed == 0) {
```

```
    seed = (uint32) -1;
  }

  int i;

  // make first 5 random numbers and put them into the buffer
  for (i = 0; i < 5; ++i) {
    seed ^= seed << 13;
    seed ^= seed >> 17;
    seed ^= seed << 5;
    fX[i] = seed * (1.0 / (65536.0 * 65536.0));
  }

  // randomize some more
  for (i = 0; i < 20; ++i) {
    double dummy = uniform();
  }
}

void RNG::setSeedWithTime(void)
{
  setSeed(time(0));
}

int RNG::uniformInt(void)
{
  return ((int) (32767.0 * uniform()));
}

double RNG::uniform(void)
{
  long double c = (2111111111.0 * fX[3]) +
                  (1492.0 * (fX[3] = fX[2])) +
                  (1776.0 * (fX[2] = fX[1])) +
                  (5115.0 * (fX[1] = fX[0])) +
                  fX[4];
  fX[4] = floorl(c);
  fX[0] = c - fX[4];
  fX[4] = fX[4] * (1.0 / (65536.0 * 65536.0));

  return fX[0];
}
```

A.5 Klasse Chrono

A.5.1 chrono.h

```
// ------------------------------
// Filename      : chrono.h
// Author        : Sven Maerivoet
// Last modified : 20/04/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#ifdef _WIN32
  #include <windows.h>
#endif

#include <string>
#include <time.h>

// string-streams are not always compatible
// => Microsoft Visual C++ 6.0 and UNIX/Linux
//    are treated separately
#ifdef _WIN32
  #include <sstream>
#else
  #include <strstream>
  #define  ostringstream ostrstream
#endif

using namespace std;

class Chrono {
  public:
    // constructor
    Chrono(void);

    // destructor
    ~Chrono(void);

    // methods
    void start(void);
    void stop(void);
    long getElapsedTime(void);
    static string convertToHMS(long elapsedTime);

  private:
    // utility methods
    static string convertIntToString(int x);

    // data-members
```

```
    bool fRunning;
    long fElapsedTime;
    long fStartTime;
};
```

A.5.2 chrono.cpp

```
// ------------------------------
// Filename      : chrono.cpp
// Author        : Sven Maerivoet
// Last modified : 20/04/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#ifdef _WIN32
  #include <windows.h>
#endif

#include "chrono.h"
#include <string>
#include <time.h>

// string-streams are not always compatible
// => Microsoft Visual C++ 6.0 and UNIX/Linux
//    are treated separately
#ifdef _WIN32
  #include <sstream>
#else
  #include <strstream>
  #define  ostringstream ostrstream
#endif

using namespace std;

// ------------
// class Chrono
// ------------

// -----------
// constructor
// -----------
Chrono::Chrono(void) : fRunning(false),
                       fElapsedTime(0),
                       fStartTime(0)
{
}

// -----------
```

```
// destructor
// -----------
Chrono::~Chrono(void)
{
}

// -------
// methods
// -------
void Chrono::start(void)
{
  time(&fStartTime);
  fRunning = true;
}

void Chrono::stop(void)
{
  if (fRunning) {
    time(&fElapsedTime);
    fElapsedTime = fElapsedTime - fStartTime;
    fRunning = false;
  }
}

long Chrono::getElapsedTime(void)
{
  if (fRunning) {
    time(&fElapsedTime);
    fElapsedTime = fElapsedTime - fStartTime;
  }

  return fElapsedTime;
}

string Chrono::convertToHMS(long elapsedTime)
{
  int hours, minutes, seconds;

  hours = elapsedTime / (60 * 60);
  elapsedTime = elapsedTime % (60 * 60);

  minutes = elapsedTime / 60;
  elapsedTime = elapsedTime % 60;

  seconds = elapsedTime;

  string HMS = "";
  if (hours < 10) {
    HMS += "0";
  }
  HMS += convertIntToString(hours) + ":";
```

```
  if (minutes < 10) {
    HMS += "0";
  }
  HMS += convertIntToString(minutes) + ":";

  if (seconds < 10) {
    HMS += "0";
  }
  HMS += convertIntToString(seconds);

  return HMS;
}

// ---------------
// utility methods
// ---------------
string Chrono::convertIntToString(int x)
{
  ostringstream ostr;
  ostr << x;

  // terminate stream
  #ifndef _WIN32
    // Microsoft Visual C++ 6.0 puts a strange symbol here
    ostr << ends;
  #endif

  return ostr.str();
}
```

A.6 Klasse Estimator

A.6.1 estimator.h

```
// ------------------------------
// Filename      : estimator.h
// Author        : Sven Maerivoet
// Last modified : 01/05/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#ifndef SM_ESTIMATOR_H // include-guard
#define SM_ESTIMATOR_H

#include "rng.h"

class Point3D {
  public:
    // constructors
    Point3D(void);
    Point3D(double x, double y, double z);

    // destructor
    ~Point3D(void);

    // data-members
    double fX, fY, fZ;
};

class Estimator {
  public:
    // constructor
    Estimator(void);

    // destructor
    ~Estimator(void);

    // methods
    double estimateMassOfBody(long numberOfPointsToGenerate);
    virtual double calculateEnclosingVolume(void);
    virtual Point3D generatePointInEnclosingVolume(void);
    virtual double calculateMassDensityInPoint(const Point3D& p);

  protected:
    RNG fRNG;
};

#endif
```

A.6.2 estimator.cpp

```
// ------------------------------
// Filename      : estimator.cpp
// Author        : Sven Maerivoet
// Last modified : 01/05/2001
// Target        : C++
//
// All rights reserved.
// ------------------------------

#include "estimator.h"

// -------------
// class Point3D
// -------------

// ------------
// constructors
// ------------
Point3D::Point3D(void) : fX(0.0),
                         fY(0.0),
                         fZ(0.0)
{
}

Point3D::Point3D(double x, double y, double z) : fX(x),
                                                 fY(y),
                                                 fZ(z)
{
}

// ----------
// destructor
// ----------
Point3D::~Point3D(void)
{
}

// ---------------
// class Estimator
// ---------------

// -----------
// constructor
// -----------
Estimator::Estimator(void)
{
  fRNG.setSeedWithTime();
}
```

```
// ----------
// destructor
// ----------
Estimator::~Estimator(void)
{
}

// -------
// methods
// -------
double Estimator::estimateMassOfBody(long numberOfPointsToGenerate)
{
  double massDensitySum = 0.0;

  for (long i = 0; i < numberOfPointsToGenerate; ++i) {
    massDensitySum +=
      calculateMassDensityInPoint(generatePointInEnclosingVolume());
  }

  return ((calculateEnclosingVolume() / numberOfPointsToGenerate) *
          massDensitySum);
}

double Estimator::calculateEnclosingVolume(void)
{
  return 1.0;
}

Point3D Estimator::generatePointInEnclosingVolume(void)
{
  return Point3D(0.0,0.0,0.0);
}

double Estimator::calculateMassDensityInPoint(const Point3D& p)
{
  return 0.0;
}
```

A.7 Klasse GearEstimator

A.7.1 estimate-gear.cpp

```
// ---------------------------------
// Filename      : estimate-gear.cpp
// Author        : Sven Maerivoet
// Last modified : 03/05/2001
// Target        : C++
//
// All rights reserved.
// ---------------------------------

#include "chrono.h"
#include "estimator.h"
#include "math.h"
#include <iostream.h>
#include <math.h>

using namespace std;

// METHOD 1 uses a complete enclosing cylinder
//          (from 0 to fRadius3)
// METHOD 2 uses a hollow enclosing cylinder
//          (from fRadius2 to fRadius3)
// METHOD 3 uses a half-tooth-cylinder-sector
//          (from 0 to fRadius3)
// METHOD 4 uses a hollow half-tooth-cylinder-sector
//          (from fRadius2 to fRadius3)

class GearEstimator : public Estimator {
  public:
    // some analytically computed values
    static const double kMassOfRing;

    // constructors
    GearEstimator(void);
    GearEstimator(double radius1,
                  double radius2,
                  double radius3,
                  double width,
                  double massDensityInRing,
                  double massDensityFactorInTooth,
                  int nrOfTeeth);

    // destructor
    ~GearEstimator(void);

    // methods
    double calculateEnclosingVolume(void);
    Point3D generatePointInEnclosingVolume(void);
```

```
    double calculateMassDensityInPoint(const Point3D& p);

  private:
    // utility methods
    void preCalculate(void);
    void reducePhi(double& phi) const;

    // data-members
    double fRadius1;
    double fRadius2;
    double fRadius3;
    double fWidth;
    double fMassDensityInRing;
    double fMassDensityFactorInTooth;
    int    fNrOfTeeth;
    double fPhiMax;
    double fCosPhiMax;
    double fSinPhiMax;
    double fToothSpanAngle;
};

// ---------------------------------
// some analytically computed values
// ---------------------------------
const double GearEstimator::kMassOfRing = 0.3769911;

// ------------
// constructors
// ------------
GearEstimator::GearEstimator(void) :
  Estimator(),
  fRadius1(1.0),
  fRadius2(5.0),
  fRadius3(8.0),
  fWidth(1.0),
  fMassDensityInRing(5.0 / 1000.0),
  fMassDensityFactorInTooth(0.5 / 10000.0),
  fNrOfTeeth(32)
{
  preCalculate();
}

GearEstimator::GearEstimator(double radius1,
                             double radius2,
                             double radius3,
                             double width,
                             double massDensityInRing,
                             double massDensityFactorInTooth,
                             int nrOfTeeth) :
  Estimator(),
  fRadius1(radius1),
```

```
  fRadius2(radius2),
  fRadius3(radius3),
  fWidth(width),
  fMassDensityInRing(massDensityInRing),
  fMassDensityFactorInTooth(massDensityFactorInTooth),
  fNrOfTeeth(nrOfTeeth)
{
  preCalculate();
}

// ----------
// destructor
// ----------
GearEstimator::~GearEstimator(void)
{
}

// -------
// methods
// -------
double GearEstimator::calculateEnclosingVolume(void)
{
  // the enclosing volume is a cylinder that extends
  // from -r3 to +r3 in the X-direction,
  // from -r3 to +r3 in the Y-direction and
  // from -width/2 to +width/2 in the Z-direction

  #ifdef METHOD1
    return (fWidth * Math::kPi * Math::sqr(fRadius3));
  #endif

  #ifdef METHOD2
    return ((fWidth * Math::kPi) *
            (Math::sqr(fRadius3) - Math::sqr(fRadius2)));
  #endif

  #ifdef METHOD3
    return ((fWidth * fPhiMax * Math::sqr(fRadius3)) / 2.0);
  #endif

  #ifdef METHOD4
    return ((fWidth * fPhiMax *
            (Math::sqr(fRadius3) - Math::sqr(fRadius2))) / 2.0);
  #endif
}

Point3D GearEstimator::generatePointInEnclosingVolume(void)
{
  Point3D p;
  bool ok;
```

```
#ifdef METHOD1
  do {
    p.fX = -fRadius3 + (2.0 * fRadius3 * fRNG.uniform());
    p.fY = -fRadius3 + (2.0 * fRadius3 * fRNG.uniform());
    p.fZ = -(fWidth / 2.0) + (fWidth * fRNG.uniform());

    double radius = sqrt(Math::sqr(p.fX) + Math::sqr(p.fY));
    ok = (radius <= fRadius3);
  } while (!ok);
#endif

#ifdef METHOD2
  do {
    p.fX = -fRadius3 + (2.0 * fRadius3 * fRNG.uniform());
    p.fY = -fRadius3 + (2.0 * fRadius3 * fRNG.uniform());
    p.fZ = -(fWidth / 2.0) + (fWidth * fRNG.uniform());

    double radius = sqrt(Math::sqr(p.fX) + Math::sqr(p.fY));
    ok = ((radius >= fRadius2) &&
          (radius <= fRadius3));
  } while (!ok);
#endif

#ifdef METHOD3
  do {
    p.fX = fRadius3 * fRNG.uniform();
    p.fY = (fRadius3 * fSinPhiMax) * fRNG.uniform();
    p.fZ = -(fWidth / 2.0) + (fWidth * fRNG.uniform());

    double radius = sqrt(Math::sqr(p.fX) + Math::sqr(p.fY));
    double phi = Math::arctan(p.fY,p.fX);
    ok = ((radius <= fRadius3) &&
          (phi <= fPhiMax));
  } while (!ok);
#endif

#ifdef METHOD4
  double lowerTreshold = fRadius2 * fCosPhiMax;
  do {
    p.fX = lowerTreshold +
           ((fRadius3 - lowerTreshold) * fRNG.uniform());
    p.fY = (fRadius3 * fSinPhiMax) * fRNG.uniform();
    p.fZ = -(fWidth / 2.0) + (fWidth * fRNG.uniform());

    double radius = sqrt(Math::sqr(p.fX) + Math::sqr(p.fY));
    double phi = Math::arctan(p.fY,p.fX);
    ok = ((radius >= fRadius2) &&
          (radius <= fRadius3) &&
          (phi <= fPhiMax));
  } while (!ok);
#endif
```

```
  return p;
}

double GearEstimator::calculateMassDensityInPoint(const Point3D& p)
{
  // the gear is cylindrical shaped so every radius
  // we measure for a fixed X- and Y-coordinate
  // is the same at every Z-coordinate
  double radius = sqrt(Math::sqr(p.fX) + Math::sqr(p.fY));

  if (radius < fRadius1) {
    // we are inside the inner hole of the gear
    return 0.0;
  }
  else if (radius <= fRadius2) {
    // we are in the part with a constant mass-density
    return fMassDensityInRing;
  }
  else if (radius <= fRadius3) {
    // we are in a tooth where the mass-density is linearly varying

    // find the angle at which the point lies in the gear
    double phi = Math::arctan(p.fY,p.fX);

    #ifdef METHOD1
      reducePhi(phi);
    #endif

    #ifdef METHOD2
      reducePhi(phi);
    #endif

    // determine the maximum radius for this angle
    // by finding the intersection between the line
    // with slope phi and the edge of the first tooth
    double tanPhi = tan(phi);
    double alpha = fRadius2 * fSinPhiMax;
    double beta = (fRadius2 * fCosPhiMax) - fRadius3;
    double x = -(alpha * fRadius3) / ((beta * tanPhi) - alpha);
    double maxRadius = x * sqrt(1.0 + Math::sqr(tanPhi));

    // check if the radius of the point lies within the tooth
    // (only the angle of the original point changed,
    // its distance remained the same)
    if (radius <= maxRadius) {
      // the mass-density varies linearly
      double massDensity = fMassDensityInRing;
      massDensity += fMassDensityFactorInTooth *
                     (radius - fRadius2);
      return massDensity;
```

```
    }
    else {
      return 0.0;
    }
  }
  else {
    // we are definitely outside the gear
    return 0.0;
  }
}

// ---------------
// utility methods
// ---------------
void GearEstimator::preCalculate(void)
{
  // the angle a tooth spans on the gear
  fToothSpanAngle = (2.0 * Math::kPi) / fNrOfTeeth;

  // the angle a halve tooth spans on the gear
  fPhiMax = fToothSpanAngle / 2.0;

  fCosPhiMax = cos(fPhiMax);
  fSinPhiMax = sin(fPhiMax);
}

void GearEstimator::reducePhi(double& phi) const
{
  // a tooth t begins at :
  //   phiBegin = -fPhiMax + (t * fToothSpanAngle)
  //
  // and ends at :
  //   phiEnd = phiBegin + fToothSpanAngle
  //
  // with t in {0,...,fNrOfTeeth - 1}

  // determine in which tooth the point is located
  int t = (int) floor((phi + fPhiMax) / fToothSpanAngle);
  if (t == fNrOfTeeth) {
    // wrap around
    t = 0;
  }

  // reduce phi so it lies in the first tooth
  // (from -fPhiMax to +fPhiMax)
  if (phi > ((2.0 * Math::kPi) - fPhiMax)) {
    phi = -((2.0 * Math::kPi) - phi);
  }
  phi -= (t * fToothSpanAngle);

  // convert phi so it lies in the first quadrant
```

```
  // by taking the absolute value
  // (the tooth is symmetrical with respect to the X-axis)
  phi = Math::abs(phi);

  // if necessary, clip phi
  if (phi < 0.0) {
    phi = 0.0;
  }
  else if (phi > fPhiMax) {
    phi = fPhiMax;
  }
}

// -------------
// main-function
// -------------
void main(int argc, char* argv[])
{
  const double kRadius1                  = 1.0;
  const double kRadius2                  = 5.0;
  const double kRadius3                  = 8.0;
  const double kWidth                    = 1.0;
  const double kMassDensityInRing        = (5.0 / 1000.0);
  const double kMassDensityFactorInTooth = (0.5 / 10000.0);
  const int    kNrOfTeeth                = 32;
  const int    kNrOfEstimates            = 10;
  const int    kNrOfSamplePoints         = NROFSAMPLEPOINTS;

  // set precision to 53 digits for floating-point output
  cout.precision(53);

  Estimator* gearEstimator =
    new GearEstimator(kRadius1,kRadius2,kRadius3,
                      kWidth,
                      kMassDensityInRing,
                      kMassDensityFactorInTooth,
                      kNrOfTeeth);

  cout << endl;
  cout << "#estimates              = "
       << kNrOfEstimates << endl;
  cout << "#sample-points/estimate = "
       << kNrOfSamplePoints << endl;
  cout << "enclosing volume        = "
       << gearEstimator->calculateEnclosingVolume()
       << " cm^3" << endl;
  cout << endl;

  #ifdef METHOD1
    cout << "Using method 1 for computation..." << endl;
  #endif
```

```
#ifdef METHOD2
  cout << "Using method 2 for computation..." << endl;
#endif

#ifdef METHOD3
  cout << "Using method 3 for computation..." << endl;
#endif

#ifdef METHOD4
  cout << "Using method 4 for computation..." << endl;
#endif

cout << endl;

Chrono overallChrono;
overallChrono.start();

double averageMass = 0.0;
double estimatedMass;
for (int estimateNr = 0;
     estimateNr < kNrOfEstimates;
     ++estimateNr) {

  Chrono specificChrono;
  specificChrono.start();

  #ifdef METHOD1
    estimatedMass =
      gearEstimator->estimateMassOfBody(kNrOfSamplePoints);
  #endif

  #ifdef METHOD2
    estimatedMass =
      GearEstimator::kMassOfRing +
      gearEstimator->estimateMassOfBody(kNrOfSamplePoints);
  #endif

  #ifdef METHOD3
    estimatedMass =
      (2.0 * kNrOfTeeth) *
      gearEstimator->estimateMassOfBody(kNrOfSamplePoints);
  #endif

  #ifdef METHOD4
    estimatedMass =
      GearEstimator::kMassOfRing +
      (2.0 * kNrOfTeeth) *
      gearEstimator->estimateMassOfBody(kNrOfSamplePoints);
  #endif
```

```
    specificChrono.stop();
    long estimationTimeNeeded = specificChrono.getElapsedTime();

    cout << "\tm[" << estimateNr << "] = "
         << estimatedMass << " kg" << endl;
    cout << "\t\ttimed needed = "
         << Chrono::convertToHMS(estimationTimeNeeded)
         << " (= " << estimationTimeNeeded << " seconds)" << endl;

    averageMass += estimatedMass;
  }
  averageMass /= kNrOfEstimates;

  overallChrono.stop();
  long overallTimeNeeded = overallChrono.getElapsedTime();

  cout << endl;
  cout << "average mass = " << averageMass << " kg" << endl;
  cout << "timed needed = "
       << Chrono::convertToHMS(overallTimeNeeded)
       << " (= " << overallTimeNeeded << " seconds)" << endl;
  cout << endl;
}
```

Referenties

[Fog93] Agner Fog. *Pseudo random number generators*. Webextract, maart 1993. (URL : `http://www.agner.org/random/`).

[Hew89] Paul G. Hewitt. Properties of matter - solids. Uit *Conceptual Physics*, hoofdstuk 11, pagina 196. Harper Collins Publishers, 6e editie, 1989.

[LS99a] John Liu en Murray R. Spiegel. Formulas from Vector Analysis. Uit *Mathematical Handbook of Formulas and Tables*, Schaum's Outline Series, hoofdstuk 20, pagina's 122,124–126. McGraw-Hill, 2e editie, 1999.

[LS99b] John Liu en Murray R. Spiegel. Geometric Formulas. Uit *Mathematical Handbook of Formulas and Tables*, Schaum's Outline Series, hoofdstuk 7, pagina 16. McGraw-Hill, 2e editie, 1999.

[Mae01] Sven Maerivoet. *Het gebruik van microscopische verkeerssimulatie bij een onderzoek naar de fileproblematiek op de Antwerpse ring*. Licentiaatsthesis, UIA - Universitaire Instelling Antwerpen, 2001. Hoofdstuk 7, paragraaf 7.1.2, pagina 135 (URL : `http://mitrasim2000.dyns.cx`).

[Rou94] Peter Rousseeuw. Beschrijvende statistiek. Uit *Elementen van kanstheorie en statistiek*, hoofdstuk 5, pagina's BS.4,BS.6. UIA - Universitaire Instelling Antwerpen, 1994.

[Str97] Bjarne Stroustrup. Random Numbers. Uit *The C++ Programming Language*, hoofdstuk 22, pagina's 685–686. Addison-Wesley, 3e editie, 1997.

[Ver01] Frans Verbeure. Toepassingen van de Monte Carlo Methode. Uit *Meten en simuleren met computers*, hoofdstuk 3, pagina 29. UIA - Universitaire Instelling Antwerpen, 2001.