Luku 1

Johdanto

1.1 Mallit ja simulointi

Simulointi ja mallit liittyvät läheisesti yhteen. Simulointi tarkoittaa pohjimmiltaan simuloitavan systeemin tai ilmiön *jäljittelyä*. Tätä varten tarvitaan *malli*: systeemi, joka ainakin oleellisilta piirteiltään käyttäytyy kuten alkuperäinen systeemi mutta on helpommin käsiteltävissä. Malli voi olla fyysinen analogia (kuten pienoismallit) tai systeemin abstraktio (matemaattinen malli, prosessin vuokaavio). Simuloinnissa alkuperäisen systeemin toiminnasta pyritään saamaan tietoa havannoimalla mallia, johon vaikuttavia tekijöitä varioidaan. Se, mitä kysymyksiä simuloinnilla halutaan selvittää, vaikuttaa mm. siihen, mitkä piirteet ovat mallittamisen kannalta oleellisia. Simuloinnin tavoite onkin kiinnitettävä ennen kuin mallia aletaan edes rakentaa.

Matemaattiset mallit voidaan jakaa karkeasti kahteen luokkaan, analyyttisiin ja numeerisiin. Analyyttisissä malleissa haluttu ominaisuus voidaan ilmaista suljettuna lausekkeena mallin tunnetuista ominaisuuksista. Esimerkkejä voisivat olla vaikka differentiaaliyhtälömallit, joille ratkaisu tunnetaan analyyttisesti. Samoin mm. yksinkertaiset jonomallit ovat usein analyyttisiä (siitä huolimatta että ne ovat stokastisia eli sisältävät satunnaisprosesseja). Muut kuin analyyttiset mallit kuuluvat numeeristen mallien luokkaan (kuten differentiaaliyhtälöt, joille ei tunneta ratkaisua suljetussa muodossa). Numeerisia malleja kutsutaan deterministisiksi, jos ne eivät sisällä satunnaismuuttujia, muussa tapauksessa puhutaan stokastisista tai Monte Carlo malleista. Stokastisella mallilla kuvattavan ilmiön ei välttämättä tarvitse olla itsessään stokastinen. Tästä klassinen esimerkki on Monte Carlo integrointi, johon palataan myöhemmin.

Tässä kurssissa keskitytään simulointiin stokastisten mallien avulla systeemeille, jotka itsessään sisältävät stokastisia komponentteja ja ovat luonteeltaan dynaamisia. Tyypillisesti on kyse systeemeistä, joissa satunnaisin väliajoin esiintyy erilaisia tapahtumia (satunnaisessa järjestyksessä). Näitä kutsutaan yleensä tapahtumapohjaisiksi (event based) tai diskreettiaikaisiksi (discrete time) malleiksi. Useimmat logistiikan, tietokonetekniikan, tietoliikenteen jne mallit ovat tämän tyyppisiä.

Onnistunut Monte Carlo simulointi edellyttää, että a) malli, kaikkine stokastisine parametreineen, kuvaa alkuperästä systeemiä riittävän tarkasti, b) mallin tietokonetoteutus voidaan tehdä luotettavasti ja kohtuullisella panostuksella ja b) simuloinnin (stokastiset) tulokset saadaan riittävällä tarkkuudella käytettävissä olevilla resursseilla ja tarkkuutta voidaan estimoida.

1.2 Esimerkki [Mitrani]

Huoltoasemayrittäjä harkitsee autonpesuaseman hankintaa. Käytettävissä on kaksi vaihtoehtoa, halpa ja hidas, sekä kallis ja nopea. Simuloinnilla halutaan selvittää kummankin vaihtoehdon kannattavuus, jotta voidaan päättää kumpi, jos kumpikaan, vaihtoehdoista tulee kyseeseen.

Primäärinen suure, joka kiinnostaa yrittäjää on tuotto P aikayksikköä kohden, joka kummallekin vaihtoehdolle on muotoa $P = aU - b$, missä b vastaa kiinteitä kuluja (korot, kuoletukset) ja a käyttökatetta (pesumaksut - muuttuvat kulut) aikayksikköä kohti. U on koneen keskimääräinen käyttöaste. a ja b tunnetaan kummallekin vaihtoehdolle. Simuloinnin osalle jää selvittää U.

Käyttöasteeseen vaikuttavia tekijöitä ovat potentiaalisten asiakkaiden määrä, josta yrittäjällä on empiiristä tietoa (odotettavissa keskimäärin n potentiaalista asiakasta aikayksikköä kohti), sekä maksimaalinen jonon pituus (vain M autoa mahtuu jonottamaan kerralla). Samoin tiedossa on yksittäisen pesun keston jakauma kummallekin koneelle. Käyttöasteen määräämiseksi on nyt selvitettävä autojen määrä pesupaikalla ajan funktiona, erityisesti ajat, jolloin asema on tyhjä.

Systeemin (aseman) tilaa riittää kuvaamaan yksi muuttuja, $N = N(t)$, autojen määrä asemalla hetkellä t. Tilaan vaikuttavia tapahtumia on kahdenlaisia, autojen saapumisia ja autojen lähtemisiä, jotka tapahtuvat ajanhetkillä t_{a_i} ja t_{d_j}. Kun nämä ajat tunnetaan, $N(t)$, ja T_0, voidaan määrätä (olettaen alkutila tunnetuksi). Aikoja ei kuitenkaan voida määrätä suoraviivaisesti kerralla, koska ne riippuvat toisistaan ja systeemin tilasta N.

Käytännössä menetellään seuraavasti: kummankin tyyppisille tapahtumille (tulo ja lähtö) varataan muuttuja (AT, DT), joka kertoo seuraavan k.o. tapahtuman ajankohdan. Lisäksi ylläpidetään systeemin tilaa (N) ja systeemin aikaa (t). Olkoon systeemi tilassa $(t, N(t))$. Nyt tarkistetaan kumpi muuttujista AT ja DT on pienempi. Näin määräytyy seuraava tapahtuma. Olkoon se uuden asiakkaan tulo. Tällöin lisätään N:n arvoa yhdellä (ellei N ole jo ylärajallaan $(M + 1)$), asetetaan $t = AT$ ja generoidaan uusi AT käyttäen tunnettua tuloaikajakaumaa. Jos asema oli tyhjä ($N = 0$) ennen tapahtumaa, on päivitettävä tyhjänäoloaikaa keräävää muuttujaa. Lisäksi on generoitava aika, jolloin tyhjään systeemiin saapunut auto poistuu (DT ei ole määritelty tyhjälle asemalle). Tämä saadaan arpomalla pesun kestoaika (satunnaismuuttuja, jonka jakauma tunnetaan) ja lisäämällä se T:hen. Vastaavasti, jos tapahtuma on lähtö, vähennetään N:n arvoa yhdellä, asetetaan $t = DT$ ja generoidaan uusi DT. Tietenkin mikäli asema tyhjenee DT ei ole relevantti. Tämä voidaan huomioida asettamalla DT käytännössä äärettömäksi (vähintään suuremmaksi kuin simuloinnin kesto).

Olkoon simuloinnin kesto T ja aseman tyhjänä oloaika T_0. Tällöin käyttöaste on $U = 1-(T_0/T)$. Koska aseman kuormitus riippuu asiakkaisen (satunnaisesta) tulosta ja palvelun satunnaisesta kestosta, myös U on satunnaisluku. Tällöin yksittäisen U:n arvon nojalla ei voi tehdä johtopäätöksiä. Rationaalinen päätöksenteko edellyttää, että U:n jakauma tunnetaan, vähintään tarvitaan estimaatti U:n odotusarvolle ja luottamusvälit k.o. estimaatille. Nämä voidaan saada vain toistamalla simulointi riittävän monta kertaa (eri satunnaislukuja käyttäen) ja laskemalla tuloksen keskiarvo ja hajonta. Yleensä 'riittävä' määrä on varsin suuri ja tärkeä osa simuloinnin tilastollista aspektia onkin se, miten simulointi tulisi järjestää, jotta mahdollisimman pieni työ antaisi halutun tarkkuuden.

1.3 Peukalosääntöjä

Simuloinnin käyttöön voidaan antaa monia ohjeita. Tärkeimmät aspektit voidaan koota esimerkiksi seuraavasti.

Älä simuloi ellei ole pakko. Käytä mielummin analyyttistä tai ainakin determinististä numeerista lähestymistapaa jos mahdollista.

Muodosta malli huolellisesti. Pidä mielessä halutun tuloksen käyttötarkoitus. Oikea vastaus väärään ongelmaan on hyödytön, samoin väärin muotoiltu vastaus. Esimerkiksi padon suunnittelussa joen keskikorkeus sadekaudella tai tulvan aikana ei ole riittävä tieto vaikka sille tunnetaisiin jakaumat ja luottamusvälit.

Validoi malli ja verifioi ohjelmisto. Mallissa tehdyt oletukset on pystyttävä todentamaan. Simuloitaessa olemassaolevaa systeemiä tämä voidaan usein tehdä kokeellisesti. Koodin verifiointi voi olla monimutkainen ongelma, jossa auttaa usein testaus erikoistapauksille, joille tulos on muuta kautta tunnettu.

Arvioi tuloksen tarkkuus. Paitsi että tuloksen luottamusväli on aina määrättävä, myös tuloksen riippuvuus mallin parametreista (syöttötietojen jakaumat) on selvitettävä.

Hanki asiantuntemus sovellusongelmasta. Simulointi on geneerinen menetelmä, jota voi soveltaa monilla aloilla. Simuloinnin expertti ei voi itse hallita kaikkia potentiaalisia sovellusaloja, joten hän on oltava valmis etsimään informatiota sieltä, mistä sitä kulloinkin on saatavissa.

Viimeiseksi, ennenkuin vastaat kysymykseen simuloinnilla, varmistu että ymmärrät kysymyksen.

1.4 Simulointimallien luokittelu

1.4.1 Monte Carlo mallit

Monte Carlo malleilla tarkoitetaan tässä yhteydessä tilastollista simulointia, jolla pyritään selvittämään staattisen prosessin tilastollisia ominaisuuksia. Tyypillisesti määräämään systeemin vasteen jakauma kun syötteen jakauma tunnetaan. Yksinkertaisin esimerkki on Monte Carlo integrointi, jossa systeemi on täysin deterministinen ja kuvaa x:n $f(x)$:lle. Integraali $\int_A f(x)dx$ voidaan nyt määrätä, jos oletetaan, että x on A:ssa tasan jakautunut satunnaismuuttuja ja määrätään $f(x)$:n odotusarvo $E(f(x))$. Normeeraamalla tämä A:n mitalla, saadaan integraalin arvo.

1.4.2 Jatkuva simulointi

Jatkuvassa simuloinnissa tarkastellaan dynaamista systeemiä, jonka dynamiikkaa hallitsevat differentiaali- tai osittaisdifferentiaaliyhtälöt. Jatkuva-aikaisen simulaattorin runko rakentuu differentiaaliyhtälöiden numeerisen ratkaisumenetelmän pohjalle. Ongelman tilastollinen aspekti liittyy yleensä tehtävän dataan, alkuehtoihin, kertoimiin ja oikeaan puoleen, joita ei tunneta tarkasti vaan satunnaismuuttujina, joiden jakauma oletetaan tunnetuksi.

1.4.3 Tapahtumapohjainen simulointi

Tapahtumapohjaisessa tai diskreettiaikaisessa simuloinnissa systeemi koostuu osista, joilla on yleensä äärellinen määrä toiminnallisia tiloja. Tilojen väliset muutokset ovat epäjatku-

via, jolloin systeemin osa siirtyy tilasta toiseen jollakin ajan hetkellä. Muutosaikojen välissä ei tapahdu mitään mielenkiintoista, joten mallittamiseen riittää määrätä transitioajat ja niihin liittyvät tapahtumat. Tämä lähestymistapa soveltuu mm. jonomalleihin, erilaisiin client-server systeemeihin jne. Stokastisuus tulee siitä, että eri tapahtumia generoituu satunnaisin väliajoin.

1.4.4 Yhdistetty simulointi

Yhdistetystä simuloinnista puhutaan kun systeemissä on sekä jatkuva-aikaisia että tapahtumapohjaisia piirteitä. Jatkuva aikaisuuden takia systeemiä on simuloitava myös tapahtumien välillä. Toisaalta diskreettiaikaiset tapahtumat aiheuttavat sen, ettei perinteinen differentiaaliyhtälöiden ratkaisu ole yksin riittävä työkalu. Yksinkertaisena esimerkkinä voidaan ajatella vaikka virastotalon ilmastointia. Lämpötilan kulkua voidaan mallittaa (osittais)differentiaaliyhtälöillä kun ulkoilman lämpötila, lämmönlähteet (lämmitys, tietokoneet) jne tunnetaan. Ilmastointia ja lämmitystä säädellään sen mukaan onko talo miehitetty vai ei ((virka-)aikaan sidottu tapahtuma), toisaalta säätösysteemi kytkee ilmastoinnin tai lämmityksen päälle termostaatin antaman informaation mukaan (jatkuvasti muuttuvaan tilaan sidottu tapahtuma).

1.5 Simulointimallinnuksen vaiheet

Riippumatta simuloitavan prosessin ja simulointimallin luonteesta simulointikokeen läpivieminen on monivaiheinen prosessi, joka tulee hahmottaa kokonaisuutena ennenkuin yksittäisiin vaiheisiin panostetaan liikaa resursseja. Jako osavaiheisiin voidaan esittää esim. seuraavasti:

- Systeemin identifiointi ja rajaaminen
- Systeemin osien ja niiden vuorovaikutuksen mallintaminen
- Mallin numeeristen suureiden määrittäminen
- Mallin koodaaminen
- Ohjelman verifiointi
- Mallin validointi
- Tuotantoajot
- Tulosten analysointi

Tarkastellaan nyt alustavasti edellä esitettyä pesuasemamallia tavoiteena tunnistaa mallituksen eri vaiheita ja niihin liittyviä kysymyksiä.

Ensimmäinen vaihe on systeemin identifiointi ja rajaaminen. Esimerkissä rajoituttiin tarkastelemaan pesuasemaa ja sen edessä olevaa jonotustilaa sekä potentiaalisten asiakkaiden (homogeenista) joukkoa. Tässä on tehty valinta olla huomioimatta pesuaseman toimintaympäristöä ja sen vaikutuksia. Esimerkiksi osa potentiaalisista asiakkaista saattaa haluta myös tankata. Tällöin asiakas, joka saapuessaan havaitsee jonon liian pitkäksi valitsee tankkauksen ja palaa sen jälkeen tarkistamaan jonotustilanteen.

Kun systeemin osat on kiinnitetty (asema, jono, asiakkaat), on yksittäisten osien toiminta ja keskinäiset interaktiot mallitettava. Pesun vaatima aika saadaan valmistajalta. Tähän on ehkä lisättävä auton sisään-/ulosajoon vaadittava aika, joka vaihdellee satunnaisesti. Myös yksikkökustannukset on arvioitava. Asiakkaiden käyttäytymiselle on valittu hyvin yksinkertainen malli. Potentiaalinen asiakas liittyy jonoon aina jos on tilaa, riippumatta jonon tai jonotusajan pituudesta. Toisaalta voi olla vaikeaa arvioida asiakkaiden kärsivällisyyden jakaumaa, joten yksinkertaistus on ainakin aluksi paikallaan. Vielä on kiinnitettävä malli, joka ennustaa potentiaalisten asiakkaiden tulon asemalle. Tämä on yhteinen ongelma kaikille jonomalleille ja standardiratkaisut tunnetaan hyvin.

Seuravaksi on kiinnitettävä mallin numeeriset suureet, tässä tapauksessa lähinnä asiakkaiden määrä ajanyksikköä kohti, mahdollisesti ajan (kellonaika, viikonpäivä, vuodenaika, tms.) funktiona. Tämä on yleensä vaativa empiirinen vaihe jo silloin kun mallitetaan toimivaa systeemiä. Nyt kun pesulaitetta ei vielä ole, todellisten asiakkaiden havainnointi ei ole mahdollista vaan on turvauduttava esimerkiksi arvioimaan tankkaajien määrää ja oletettava, että näistä tietty murto-osa haluaisi myös pestä autonsa. Jos voitaisiin havainnoida toimivaa systeemiä (vaikka tilanteessa, jossa mietitään korvataanko hidas vanha asema uudella nopealla), olisi mahdollista pyrkiä mittaamaan vaikuttaako jonon pituus tulevien asiakkaiden määrään (asiakkaita tulee suhteessa useammin, kun jono on lyhyt). Tällöin mallia olisi harkittava uudelleen.

Mallin koodaaminen on itsessään monivaiheinen prosessi, johon ei esimerkissä vielä puututtu. Ohjelman verfiointivaiheessa voidaan pyrkiä esimerkiksi yksinkertaistamaan mallissa käytettyjä jakaumia niin, että tehtävälle löytyy analyyttinen ratkaisu. Samoin voidaan ratkoa malliongelmia, jotka löytyvät kirjallisuudesta.

Vasta kun koodi on testattu on syytä aloittaa testit mallin todellisilla parametreilla ja arvioida mallin käyttäytymistä - aluksi laadullisesti (vastaako prosessin kulku odotettua skenariota) ja sitten määrällisesti, jos kokeellista dataa on saatavilla. Esimerkissämme vertailu mitattuun tuloksiin ei ole mahdollinen, joten mallin luotettavuutta on arvioitava muilla keinoin. Esimerkiksi voi olla valaisevaa testata mallin tulosten riippuvuutta syöttötiedoista (miten asiakkaiden tulojakauman, pesun keston jakauman jne muutokset näkyvät tuloksissa). Näin voidaan arvioida, mitkä epävarmuustekijät ovat kriittisimpiä tulosten luotettavuuden kannalta. Kriittisten osien mallitusta voidaan tässä vaiheessa arvioida uudelleen.

Kun mallin toimintaan on saatu hyvä tuntuma, voidaan tehdä ensimmäiset tuotantoajot. Toisin sanoen simuloida molempia systeemivariantteja ja vertaila tuloksia. Tämä edellyttää yleensä suurta määrää toistoja, jotta tulosten hajonnan vaikutus voidaan arvioida ja alustavat johtopäätökset tehdä. Tässä vaiheessa on syytä palata mallin epävarmuustekijöihin ja tarkistaa, mitkä niistä mahdollisesti vaikuttavat johtopäätöksiin. Tarvittaessa on mallia tarkennettava kriittisiltä osin, esimerkiksi mallittamalla jonon pituuden vaikutus pesuhalukkuuteen.

Luku 2

Monte Carlo menetelmät ja satunnaisluvut

2.1 Monte Carlo

2.1.1 Buffonin neula

Buffonin neula on simulointikirjallisuudessa usein käytetty esimerkki (fysikaalisesta) stokastisesta prosessista periaatteessa deterministisen suureen määräämiseksi. Kyseessä on Buffonin herttuan v. 1733 esittämästä tavasta arvioida π:n arvoa. Menetelmä on lyhyesti seuraava: käytettävissä on neula, jonka pituus on l sekä taso, johon on merkitty yhdensuuntaisia viivoja, joiden keskinäinen etäisyys on d. Neula pudotetaan 'satunnaisesti' tasolle ja todetaan leikkaako neula tasoon merkittyjä viivoja. Jos $d \geq l$ voidaan osoittaa, että todennäköisyys sille, että neula leikkaa viivaa on $p = 2l/(\pi d)$. Kun l ja d olivat tunnettuja, määräämällä p saadaan arvo π:lle. Käytännössä tietysti p:tä ei voida määrätä tarkasti, jos suoritetaan vain äärellinen määrä kokeita.

Olkoon $\hat{p}$ kokeellisesti saatu arvo p:lle ($\hat{p}$ = osumien määrä/ toistojen määrä). Jos toistojen määrä kiinnitetään (n) ja koesarja uusitaan, $\hat{p}$:n arvo vaihtelee satunnaisesti, sillä itseasiassa $\hat{p}$ edustaa satunnaismuuttujaa, jolla on tietty (n:stä riippuva) todennäköisyysjakauma. Ellei tätä jakaumaa tunneta, ei voida sanoa juuri mitään siitä miten luotettava estimaatti $\hat{p}$ on p:lle ja miten tarkka arvio edelleen saadaan π:lle.

Käytännössä edellä kuvattu menettely on hankala toteuttaa. Miten taata että neula todella tippuu satunnaiseen paikkaan ja satunnaiseen suuntaan. Joka tapauksessa prosessi on hyvin hidas, jos neulaa liikutellaan mekaanisesti. Tämän takia onkin syytä tarkastella, miten koe voitaisiin korvata numeerisella simuloinnilla.

Neulan paikka ja asento (suhteessa viivastoon) määräytyy kahdesta parametrista, neulan painopisteen etäisyydestä Y lähimpään viivaan ja neulan terävästä kulmasta θ viivaston suhteen. Nyt, jos neula asettuu tasolle satunnaisesti, Y saa tasaisesti arvoja väliltä $0 \leq Y \leq d/2$. Samoin orientaatio vaihtelee tasaisesti välillä $(0, \pi/2)$.

On helppo todeta, että neula risteää viivan kanssa, jos ja vain jos $Y \leq (l/2)\sin\theta$. Tällöin voidaan johtaa seuraava algoritmi kokeen simuloimiseksi:

1. Aseta $p = 0$, kiinnitä n, d, l.

2. Toista n kertaa seuraava:

3. Valitse satunnaisluvut Y ja θ, Y tasan jakautunut välille $(0, d/2)$, θ tasan jakautunut välille $(0, \pi/2)$. Jos $Y \leq (l/2)\sin(\theta)$, aseta $p = p + 1$.

4. Aseta $\hat{p} = p/n$.

5. Estimoi $\hat{\pi} = 2l/(d\hat{p})$.

2.1.2 Integrointi

Toinen perinteinen esimerkki Monte Carlo menetelmistä on integrointi eli (yleensä moniulotteisen) integraalin arvon approksimointi tilastollisesti. Tarkastellaan yksiulotteista esimerkkiä: Määrää integraalin

$$I = \int_a^b f(x)dx \tag{2.1}$$

arvo. Oletetaan, että f on ei-negatiivinen ja rajoitettu, $0 \leq f(x) \leq c$. Olkoon nyt (x, y) mielivaltainen piste joukosta $[a, b] \times [0, c]$. Todennäköisyys sille, että $y \leq f(x)$ on $I/(b-a)c$. Tämä johtaa seuraavaan menetelmään I:n arvioimiseksi. Arvotaan n pistettä joukosta $[a, b] \times [0, c]$ (käytännössä n paria satunnaislukuja x ja y, siten että x:t ovat tasan jakautuneita välille $[a, b]$ ja y:t välille $[0, c]$). Lasketaan sellaisten parien määrä P, joille $y \leq f(x)$ sekä asetetaan $\hat{p} = P/n$. Arvio I:lle on $\hat{I} = \hat{p}c(b-a)$.

Helposti havaitaan, että $\hat{I}$:n arvo on satunnainen (vaihdellen pahimmillaan 0:n ja $c(b-a)$:n välillä). $\hat{I}$:n arvo on käyttökelpoinen vain, jos satunnaisuuden laatu ($\hat{I}$:n jakauma) tunnetaan.

Tarkastellaan nyt tarkemmin $\hat{p}$:tä satunnaismuuttujana. Voimme kirjoittaa $\hat{p} = \sum_{i=1}^{n} z_i/n$, missä z_i on satunnaismuuttuja, joka saa arvon 1 jos $y_i \leq f(x_i)$, muuten arvon 0. Muuttujien z_i jakaumat tunnetaan. Kyse on binomijakaumasta, jossa z_i:n odotusarvo on $p = I/(b-a)c$. Koska z_i:t ovat riippumattomia, on $\hat{p}$:n odotusarvo $np/n = p$. Edelleen z_i:lle tunnetaan varianssi, joka on $Var(z_i) = p(1-p)$. Tämän avulla voidaan määrätä myös $\hat{p}$:n varianssi,

$$Var(\hat{p}) = \sum_{i=0}^{n} Var(z_i/n) = np(1-p)/n^2 = p(1-p)/n. \tag{2.2}$$

Tästä näemme, että mitä useampaa pistettä käytämme, sitä pienempi on satunnaisvaihtelu keskiarvoestimaatille. Kääntäen, mitä tarkempi estimaatti halutaan, sitä enemmän pisteitä on käytettävä.

Vakiintunut käsite estimaatin tarkkuuden arviointiin on luottamusväli: Haluamme löytää arvon δ siten, että $p - \delta \leq \hat{p} \leq p + \delta$ tietyllä todennäköisyydellä (esimerkiksi 99 prosentissa kaikista simuloinneista). Nyt todennäköisyyslaskennan vahva raja-arvolause sanoo, että $\hat{p}$:n jakauma konvergoi kohti normaalijakaumaa kun n kasvaa. Tällöin suurilla n voidaan olettaa, että $\hat{p}$ on likimain normaalisti jakautunut keskiarvolla p ja keskihajonnalla $\sigma = (p(1-p)/n)^{1/2}$. Tästä seuraa edelleen, että $(\hat{p} - p)/\sigma$ noudattaa likimain normeerattua normaalijakaumaa. Nyt kun määrätään arvo $\delta_{.01}$, jolle pätee $P(|z| > \delta_{.01}) = .01$, kun z on normeerattu normaalisti jakautunut satunnaismuuttuja, voidaan δ:n arvo määrätä: $\hat{p}$ on p:stä korkeintaan $\sigma\delta_{.01}$:n päässä 99 prosentin todennäköisyydellä.

Koska σ on verrannollinen $n^{-1/2}$:een havaitaan, että estimaatin tarkkuuden kaksinkertaistaminen (luottamusvälin puolittaminen) edellyttää pisteiden määrän nelinkertaistamista. Näin ollen menetelmällä on vaikea päästä suureen tarkkuuteen. Toisaalta estimaatin

tarkkuus ei millään tavoin riipu integraalin dimensiosta. Tästä syystä menetelmä on suosittu erityisesti tapauksissa, joissa on integroitava hyvin moniulotteisissa joukoissa, joissa perinteiset menetelmät helposti ovat hyvin kalliita käyttää.

Edellä kuvattu tapa ei tietenkään ainut eikä aina paraskaan tapa soveltaa Monte Carlo menetelmää. Yhtä luonnollinen tapa integraalin estimoimiseksi on arvioida suoraan $f(x)$:n odotusarvoa kun x on tasan jakautunut välille $[a, b]$. Tällöin tarvitaan vain yksi satunnaismuuttuja havaintopistettä kohti, joten simulointi voidaan suorittaa tehokkaammin. Toisaalta menetelmän tarkkuus tulee nyt riippumaan $f(x)$:n varianssista, jolle on laskettava estimaatti erikseen.

2.2 Satunnaislukujen generointi

Edellä esitetty Monte Carlo integrointi perustui mahdollisuuteen valita (tasan jakautuneita) satunnaisia arvoja muuttujille x ja y. Miten tämä toteutetaan käytännössä? Jos integrointi tehdään ohjelmallisesti, on myös satunnaisluvut generoitava satunnaisesti. Tässä on jo sinällään käsitteellinen ristiriita, kuten John v. Neumann sanoi:

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

Lisäksi on määriteltävä, mitä tarkoitetaan satunnaisluvulla. Missä mielessä esimerkiksi 175 on satunnaisluku?

Simuloitaessa tarvitsemme jonon lukuja, jotka ovat (näennäisesti) keskenään riippumattomia ja joiden arvot noudattavat jotain haluttua todennäköisyysjakaumaa. Algoritmeja, jotka tuottavat tällaisia lukusarjoja kutsutaan satunnaislukugeneraattoreiksi (tai pseudosatunnaislukugeneraattoreiksi). Näennäisellä riippumattomuudella tarkoitetaan sitä, että käytetyt tilastollista riippuvuutta indikoivat testit eivät näe lukujen välillä systemaattista riippuvuutta. Se, mitä testejä generaattorille tulee tehdä, riippuu osin simulointiongelmasta. Satunnaislukujono, joka yhdelle tehtävälle toimii hyvin, voi toisessa ongelmassa johtaa täysin vääriin tuloksiin.

Satunnaislukujen generoinnissa voidaan erottaa kaksi alaongelmaa. Toinen on tasan jakautuneiden satunnaislukujen generointi (välille $(0, 1)$), toinen on halutun todennäköisyysjakauman mukaisten satunnaislukujen generointi. Tulemme näkemään, että jälkimmäisen ongelman ratkaisu edellyttää ensimmäisen tehtävän hallintaa.

2.2.1 Keskineliömenetelmä

Tarkastellaan aluksi klassista (huonoa) ad hoc menetelmää, jonka esitti J. v. Neumann vuonna 1946. Kyseessä on niin sanottu keskineliömenetelmä, joka toimii seuraavasti. Olkoon annettu k-numeroinen luku (jossakin kannassa), esimerkiksi 1234. Kerrotaan luku itsellään, jolloin saadaan $2k$-numeroinen luku (täyttämällä alkuun nollia tarvittaessa). Tässä tapauksessa $1234 * 1234 = 01522756$. Otetaan tästä luvusta k keskimmäistä numeroa, joista muodostuu uusi k-numeroinen luku, 5227. Prosessia voidaan jatkaa, jolloin $5227 * 5227 = 27,3215,29$ ja niin edespäin. Näin saadaan päättymätön sarja lukuja, jotka eivät näytä mitenkään riippuvan toisistaan. Menetelmä soveltuu mihin tahansa lukujärjestelmään, erityisesti myös binäärijärjestelmään, joten se on varsin helppo toteuttaa.

Menetelmä on kuitenkin huono. On helppo todeta, että menetelmä päätyy johonkin lukusykliin. (Nelinumeroisia lukuja on 10.000 kappaletta, joten viimeistään 10.000 askeleen

jälkeen jokin toistuu ja prosessi alkaa toistaa itseään.) Ongelma on siinä, että käytännössä syklit löytyvät paljon nopeammin ja ovat hyvin lyhyitä. Samoin tietyt poikkeusarvot (esimerkiksi alle 100:n olevat luvut tai 100:n monikerrat) johtavat 'ei satunnaisiin' lukusarjoihin.

Satunnaislukujen generointi ei siis onnistu aivan helposti. Tästä on lisää esimerkkejä mm. Knuth, vol 2, s.4, jossa esitetään 'supersatunnainen' algoritmi, joka tekijänsä yllätykseksi oli täysin susi.

2.2.2 Lehmer generaattorit

Yleisesti käytetyin tapa satunnaislukujen generointiin perustuu D. Lehmerin vuonna 1948 esittämään menetelmään (jonka hän implementoi ENIACille). Menetelmää kutsutaan lineaarisen kongruenssin menetelmäksi, ja siinä on kolme valittavaa parametria, kerroin a, lisäys c ja kantaluku m. Lisäksi tarvitaan siemenluku X_0. Menetelmä generoi jonon X_i käyttäen sääntöä

$$X_{n+1} = (aX_n + c) \mod m, \qquad n \geq 0.$$

Havaitaan, että X_n:t ovat väliltä $0, m-1$, joten normeeraamalla $m-1$:llä saamme välille $[0,1]$ jakautuneita lukuja.

Koska X_{n+1} riippuu vain X_n:stä ja molemmat voivat saada vain m erilaista arvoa, viimeistään m askeleen jälkeen jono päätyy sykliin. Jos kerrointen valinnassa ei olla huolellisia, sykli voi olla luonnollisesti paljon lyhyempi kuin m. Onneksi maksimaaliseen sykliin johtavat kertoimet osataaan karakterisoida:

Lause 1 *Lineaarisen kongruenssin menetelmä johtaa m pituiseen sykliin jos ja vain jos*

1. *$c \neq 0$ ja c:llä ja m:llä ei ole yhteisiä tekijöitä.*
2. *Jos q on m:n tekijä, on oltava $a = 1 \mod q$.*
3. *Jos m on jaollinen 4:llä, on oltava $a = 1 \mod 4$.*

Jos tarkastellaan generaattoreita, joissa $c = 0$, on helppo nähdä että täysi m alkion sykli ei ole mahdollinen (jos $X_i = 0$, $X_k = 0$ kaikille $k \geq i$). Jos m on alkuluku, on mahdollista löytää a:n arvoja siten, että syklin pituus on $m-1$. Ts. kaikki muut luvut paitsi 0 käydään läpi. Tämän ehdon toteuttavat generaattorit ovat yksinkertaisempia kuin yleiset generaattorit. Lisäksi tiedetään, että luku 0 ei esiinny tulossekvenssissä, joten satunnaisluvulla voi esimerkiksi jakaa ilman pelkoa nollalla jaosta. Tämän tyyppisestä generaattorista käytetään nimitystä 'prime modulus multiplicative congruence generator'.

Generaattoria voi yrittää nopeuttaa myös valitsemalla m siten, että jakolasku on helppo. Käytännössä asettamalla $m = 2^k$, missä k on sananpituus. Jos $c = 0$ ja $m = 2^k$, maksimaalinen syklin pituus on 2^{k-2}, jos siemenluku on pariton ja a on valittu sopivasti. Sykli ei kuitenkaan kaikilta osiltaan käyttäydy täysin satunnaisesti. Esimerkiksi viimeinen bitti muuttuu syklisesti syklin pituudella 2. Yleisemmin viimeiset j bittiä omaavat periodin 2^j.

Muutenkaan kongruenssigeneraattorin tulokset eivät voi olla täysin satunnaisia. Niille pätee mm. seuraava rajoite. Jos generaattori tuottaa lukusarjan $\{U_1, U_2, U_3, \ldots\}$ ja tarkastelemme seuraavia R^n:n pisteitä

$$\{U_1, U_2, \ldots U_n\}, \{U_1, U_2 \ldots, U_n\}, \{U_1, U_2, \ldots, U_n\}, \ldots$$

ne asettuvat avaruudessa korkeintaan $(n!m)^{(1/n)}$ yhdensuuntaiseen hypertasoon. Usein pisteet sijoittuvat huomattavasti pienempään määrään hypertasoja. Tällä voi olla haitallinen vaikutus esimerkiksi laskettaessa moniulotteisia integraaleja Monte Carlolla.

Tuntematon generaattori tulisikin aina testata ennenkuin sen arvoja käytetään todellisiin simulointeihin. Esimerkiksi alkuperäisen PC:n mukana tullut satunnaislukugeneraattori on täysin defektiivinen.

Yleisimmin käytetyt Lehmer generaattorit ovat nykyään kaikki samaa tyyppiä. Niissä on valittu $c = 0$ ja $m = 2^{31} - 1$. Tällä saavutetaan hyvä laskennallinen tehokkuus ja pitkä sykli, koska m on alkuluku. Eri generaattorit eroavat vain a:n arvojen osalta. Parhaan a:n löytämiseksi tehty laajoja testejä, joissa on vertailtu yli 267 miljoonaa eri arvoa. Viisi 'parasta' a:n arvoa olivat (Fishman, Moore, SIAM J. Sci Stat Comp, 7, ss. 24-45, (1985)) 950706376, 742938285, 1226874159, 62089911 ja 1343714438.

Käytännössä suositaan versioita, joissa a:n arvot ovat paljon pienempiä, koska tällöin generaattorien toteutus onnistuu paremmin myös korkean tason kielillä. Esimerkiksi $a = 16807$ on hyvin testattu arvo. Sen käyttöä puoltaa se, että tulo ma voidaan vielä evaluoida tarkasti käyttäen kaksoistarkkuuden liukulukuja. Siispä generaattori voidaan koodata helposti millä tahansa korkean tason kielellä, joka tukee kaksoistarkkuuden lukuja. Tehokas toteutus edellyttää yleensä jakolaskun (modulo-operaation) toteuttamista bittitason siirtojen avulla.

Lehmer-generaattoreista voidaan tehdä muunnelmia, joilla on pidempi sykli tai jotka soveltuvat paremmin normaalilla kokonaislukuaritmetiikalla käsiteltäviksi. Yhdistelmägeneraattorissa, jonka esittivät Wichmann ja Hill, otetaan useampi pienehköön alkulukuun perustuva Lehmer generaattori ja muodostetaan satunnaisluku summaamalla tulokset yhteen. Jos eri generaattorien syklit ovat keskenään jaottomia saadaan maksimissaan syklien pituuksien tulon mittainen sykli. Pienten kantalukujen käyttö toisaalta mahdollistaa standardi kokonaislukujen käytön myös välituloksille. Alkuperäinen Wichmann-Hill generaattori on

$$\begin{aligned} X_{i+1} &= 171X_i \mod 30269 \\ Y_{i+1} &= 172Y_i \mod 30307 \\ Z_{i+1} &= 170Z_i \mod 30323 \\ U_{i+1} &= \{X_{i+1}/30269 + Y_{i+1}/30303 + Z_{i+1}/30323\} \mod 1. \end{aligned}$$

Tämä tarvitsee kolme siemenlukua yhden asemasta ja on noin kolme kertaa kalliimpi kuin tavallinen Lehmer generaattori.

Toinen muunnelma on sekoitettu (suffled) generaattori. Siinä talletetaan joukko satunnaislukuja muistiin, josta niitä otetaan satunnaisessa järjestyksessä. Tätä varten on generoitava jono satunnaislukuja U (väliltä $(0, 1)$) ja toinen jono kokonaislukuja väliltä $(1, k)$, missä k on muistipaikkojen määrä. Algoritmi etenee seuraavasti. Ensin generoidaan k satunnaislukua U, jotka talletetaan k muistipaikkaan. Sitten generoidaan kokonaisluku V väliltä $(1, k)$. Haluttu satunnaisluku on muistipaikassa V, josta se välitetään eteenpäin pääohjelmaan. Lisäksi arvotaan uusi U, joka sijoitetaan edellä käytettyyn muistipaikkaan. Tämän jälkeen uudet satunnaisluvut saadaan aina määrämällä uusi V, lukemalla vastaava muistipaikka ja generoimalla sinne uusi U. Tällöin tarvitaan periaatteessa kahden luvun generointi yhtä tulosta kohti. Tätä voi kuitenkin tehostaa konstruoimalla V edellisen U:n avulla (esimerkiksi sen tiettyjen bittien avulla), jolloin lisäkustannus on vain pari operaatiota ja riittävä määrä muistipaikkoja.

2.3 Satunnaisluvut yleisestä jakaumasta

Usein tarvitaan satunnaislukuja, jotka noudattavat jotain tiettyä jakaumaa. Näitä voidaan generoida systemaattisesti käyttäen hyväksi tasan jakautuneita satunnaislukuja mikäli halutun jakauman kertymäfunktio tunnetaan.

2.3.1 Käänteistodennäköisyyden menetelmä

Oletetaan, että käytettävissä on tasan jakautuneita satunnaislukuja väliltä $(0,1)$. Haluamme muodostaa jonon satunnaislukuja, jotka noudattavat todennäköisyysjakaumaa $f(x), x \in X \subset R$. Jakaumaa (tiheysfunktiota) f vastaa kertymäfunktio $F, F(x) = \int_{-\infty}^{x} f(z)dz$ (oletetaan että $f(z) = 0$ kun $z \notin X$). Tiedetään, että F on monotonisesti kasvava funktio (aidosti pisteissä, joissa $f(x) > 0$), joka saa arvoja väliltä $[0,1]$. Jos f on funktio, F saa itse asiassa kaikki arvot. Tällöin voidaan määritellä F:n käänteisfunktio F^{-1}, joka kuvaa välin $(0,1)$ X:lle. Nyt pätee: jos (satunnaismuuttuja) U on tasan jakautunut välille $[0,1]$, noudattaa satunnaismuuttuja $F^{-1}(U)$ jakaumaa f.

Mikäli käänteisfunktio F^{-1} on esitettävissä suljetussa muodossa, on satunnaislukujen generointi jakaumasta f suoraviivainen operaatio. Esimerkkinä voimme tarkastella eksponenttijakaumaa, jonka tiheysfunktio on $f(x) = 0, x < 0$, $f(x) = \lambda e^{-\lambda x}, x \geq 0$ (missä λ on jakauman parametri. Helposti nähdään, että kertymäfunktio on muotoa $F(x) = 1 - e^{-\lambda x}$ kun $x \geq 0$. Edelleen $F^{-1}(U) = -\ln(1-U)/\lambda$, minkä avulla saadaan exponenttijakautuneita satunnaislukuja. Jos vielä huomioidaan, että $1 - U$ voidaan korvata U:lla, joka myös on tasan jakautunut ja U lasketaan generaattorilla, joka antaa aidosti positiivisia satunnaislukuja, algoritmi on erittäin suoraviivainen.

Diskreeteille todennäköisyysjakaumille F ja F^{-1} ovat porrasfunktioita ja käänteistodennäköisyyden menetelmä toimii käytännössä taulukkohakuna.

2.3.2 Eliminointimenetelmä

Usein kertymäfunktion käänteisfunktiota ei voida muodostaa eksplisiittisesti. Mikäli jakauman tiheysfunktio on rajoitettu ja kompaktikantajainen voidaan käyttää eliminointimenetelmää. Tämä on itseasiassa sukua edellä esitetylle Monte Carlo integroinnille. Oletusten mukaan tiheysfunktio f häviää jonkin välin $[a,b]$ ulkopuolella ja kyseisellä välillä pätee $0 \leq f(x) \leq c$ jollekin c. Nyt valitaan kaksi satunnaislukua x ja y siten, että x on tasan jakautunut välille $[a,b]$ ja y tasan jakautunut välille $[0,c]$. Jos nyt $f(x) > y$, hyväksytään x, muussa tapauksessa arvotaan uudet x ja y. Jokaista hyväksyttyä satunnaislukua kohti tarvitaan tasanjakautuneita satunnaislukuja keskimäärin $2c(b-a)$ kappaletta. Menetelmä on siis sitä tehokkaampi, mitä lähempänä jakauma on tasajakaumaa. (Aina pätee $c(b-a) \geq \int_a^b f(x)dx$ eron ollessa sitä suurempi mitä 'epätasaisempi' jakauma f on.)

Eliminointimenetelmää voi modifioida, jos jakauma poikkeaa huomattavasti tasajakaumasta, korvaamalla x:n tasajakauma jollakin toisella helposti generoitavalla jakaumalla g, jolle pätee $f(x) \leq cg(x)$ (sopivan pienelle c). Tällöin y:lle käytetään tasajakaumaa välillä $[0, cg(x)]$ hyväksymisehdon ollessa edelleen $f(x) > y$. Hyväksymistodennäköisyys on nyt $1/c$.

Mikäli sopivaa modifioitua jakaumaa ei löydy suoraan, voi tilannetta helpottaa osittamalla ongelma. Tämä tapahtuu siten, että jakauma f esitetään muodossa

$$f(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \cdots + \alpha_n f_n(x)$$

missä jakaumat f_i ovat paremmin approksimoitavissa yksinkertaisilla jakaumilla. Yksinkertaisimmillaan tämä voi tarkoittaa välin $[a, b]$ jakamista osaväleihin Δ_i, joiden todennäköisyydet $\alpha_i = \int_{\Delta_i} f(x)dx$ määrätään etukäteen. f_i on tällöin f:n rajoittuma välille Δ_i normeerattuna $1/\alpha_i$:llä. Jos väli Δ_i on lyhyt, voidaan f_i:tä approksimoida varsin hyvin tasajakaumalla tai jollakin muulla yksinkertaisella jakaumalla. Menetelmä on nyt seuraava: ensiksi valitaan kokonaisluku i väliltä $[1, n]$ siten, että i:n todennäköisyys on α_i. Tämän jälkeen valitaan x väliltä Δ_i noudattaen jakaumaa f_i. Tähän voidaan käyttää tarvittaessa eliminointimenetelmää, jossa välin lyhyyden ansiosta turhien pisteiden todennäköisyys on pieni.

Menetelmää voi edelleen tehostaa jakamalla kunkin f_i:n tasajakaumaan ja pieneen jäännöstermiin. Tällöin suurella todennäköisyydellä valitaan jokin tasajakaumista, jolloin varsinaisen satunnaismuuttujan konstruointi on helppoa. Vain pienellä todennäköisyydellä joudutaan jäännösjakaumiin, joissa käytetään eliminointimenetelmää, sielläkin hyvällä hyötysuhteella.

Menettelytapa voidaan laajentaa myös ei-kompaktikantajaisiin jakaumiin. Tällöin äärettömyyspisteiden ympäristöön tehdään jäännöskaistat, joissa jakaumaa arvioidaan ylöspäin esimerkiksi eksponenttijakaumalla ja sovelletaan eliminointimenetelmää.

Loppuun asti viritettynä tämä idea on viety Marsaglia, MacLaren & Bray algoritmissa normaalijakautuneiden satunnaismuuttujien generoimiseksi. Menetelmä on esitetty tarkasti mm. Knuthin kirjassa. Periaate on sama kuin yllä. Jakauma jaetaan yksinkertaisiin osiin. Symmetrian nojalla riittää osittaa vain positiiviset arvot, jotka jaetaan aluksi 1/4 mittaisiin osaväleihin. Näin jakauma pilkotaan kapeisiin yläreunastaan kaareviin nauhoihin. Kuhunkin nauhaan sovitetaan maksimaalinen suorakaide. Tästä edelleen erotetaan suorakaide, jonka pinta-ala on 1/256-osan monikerta. Vastaavasti syntyy 12 kapeaa kaistaa f_{13} - f_{24}. Nyt olkoon annettu binäärikoodattu satunnaisluku väliltä $(0, 1)$. Sen ensimmäinen (merkitsevin) bitti kiinittää generoitavan satunnaisluvun merkin. Seuraavat 8 bittiä edustavat kokonaislukua väliltä $(0, 255)$, joka joko kiinnittää yksikäsitteisesti jonkin jakaumista $f_1 - f_{12}$ tai jäännösosan $f_{13} - f_{37}$. Viimemainitun todennäköisyys on jo varsin pieni, 31/256. Mikäli valittiin jokin tasajakaumista, satunnaisluvun loppuja bittejä (kymmenennestä eteenpäin) käyttäen voidaan kiinittää haluttu tulos. Toisin sanoen 88 prosentissa tapauksista riittää yksi tasajakautunut satunnaisluku.

Jos alkuperäinen satunnaisluku on suurempi kuin 225/256 on valittava jokin alueista $f_{13} - f_{37}$. Tämä voidaan tehdä käyttäen ennalta laskettua aputaulukkoa, jossa on annettu raja-arvot ym. satunnaisluvulle. Mikäli valittiin $f_{13} - f_{24}$ riittää arpoa toinen tasan jakautunut satunnaisluku, jolla kiinnitetään lopputulos valitulta 1/4 pituiselta väliltä. Jäljelle jäävät jakaumat $f_{25} - f_{36}$, joihin sovelletaan kolmiomaisille jakaumille viritettyä eliminointimenetelmää sekä f_{37}, joka kattaa häntätermin. Tätä tarvitaan vain joka 400 kerta.

Näin saatu menetelmä on hyvin nopea mutta jo sangen monimutkainen toteuttaa. Lisäksi se edellyttää yli sadan vakion laskentaa etukäteen ja vastaavan määrän muistia. Jos nopeus ei ole kriittinen tekijä, voidaan turvautua seuravaan toteutukseltaan hyvin yksinkertaiseen algoritmiin, ns. Box-Muller(-Marsaglia) algoritmiin: generoi kaksi tasan jakautunutta satunnaislukua U_1 ja U_2. Laske $X = (-2\ln(U_1))^{1/2}\cos(2\pi U_2)$, $Y = (-2\ln(U_1))^{1/2}\sin(2\pi U_2)$. Sekä X että Y ovat normaalijakautuneita. Lisäksi ne ovat keskenään riippumattomia. Lisäkustannus on siis vain yksi logaritmi, yksi neliöjuuri ja kaksi trigonometrista funktiota kahta satunnaislukua kohti. Tosin tämäkin voi olla jo paljon verrattuna edelliseen algoritmiin.

Edellä kuvattu menetelmä ei ole mitenkään ilmeinen. Taustalla on tason normaalijakautunut pisteistö ja sen esitys napakoordinaatistossa. Olkoon (r, θ) pisteen (X, Y) esitys napakoordinaateissa. Tällöin $\theta = 2\pi U_2$ ja $r = (-2\ln(U_1))^{1/2}$. Koska U_1 oli tasan jakautunut havaitsemme, että r:n tiheysfunktio on $f(r) = e^{-r^2/2}r$. Edelleen parin (r, θ) yhteisjakauman tiheysfunktio on $f(r, \theta) = e^{-r^2/2}r/2\pi = e^{-(X^2+Y^2)/2}/2\pi$. (Tässä 'ylimääräinen' r liittyi muuttujanvaihtoon karteesisen ja napakoordinaatiston välillä). Tiheysfunktio voidaan siis ilmoittaa kahden yksiulotteisen normaalijakauman tiheysfunktion tulona.

2.4 Satunnaislukujen testaus

Koska pseudosatunnaisluvut konstruoidaan algoritmilla, ne eivät luonnollisestikaan käyttäydy täysin samoin kuin todella satunnaiset luvut. Arvioitaessa tietyn satunnaislukugeneraattorin ominaisuuksia keskeinen kysymys on määritellä ne ominaisuudet, joiden suhteen generoidun lukusarjan tulisi käyttäytyä kuten täysin satunnaisen lukujonon. Jokaista haluttua ominaisuutta kohden tulee suunnitella (toteutuskelpoinen) tilastollinen testi. Esittelemme seuraavassa muutamia testejä, joita käytetään toki myös moniin muihin tarkoituksiin.

2.4.1 Tasajakautuneisuus ja χ^2-testi

Rajoitutaan yksinkertaisuuden vuoksi tarkastelemaan tasajakautuneita satunnaislukuja väliltä $(0, 1)$. Ovatko luvut todella tasajakautuneita? Jaetaan väli d erilliseen osaan, joiden pituudet ovat p_j. Arvotaan n satunnaislukua. Nyt välille j tulisi osua keskimäärin np_j lukua. Merkitään f_j:llä välille j osuneiden lukujen määrää ja muodostetaan testisuure

$$\chi^2 = \sum_{j=1}^{d} (f_j - np_j)^2/np_j.$$

Mitä tiedämme χ^2:n jakaumasta. Kukin f_j on binomijakautunut keskiarvolla np_j. Jos np_j on suuri, binomijakaumaa voi approksimoida normaalijakaumalla. Siispä $f_j - np_j$ on likimain normaalijakautunut keskiarvolla 0 (ja varianssilla $np_j(1-p_j)$). Siispä j:s summatermi on likimain $(1-p_j)$ kertaa $N(0,1)$ jakautuneen satunnaismuuttujan neliö. Toisaalta tiedetään, että $\sum_j f_j = n$, joten summan termeistä vain $d-1$ on riippumattomia. Edelleen tekijöistä $1-p_j$ seuraa, että χ^2 käyttäytyy oleellisesti kuten $d-1$:n riippumattoman $N(0,1)$ muuttujan neliön summa. Tälle voidaan puolestaan johtaa tiheysfunktio, joka on

$$h(x) = x^{(d-3)/2}e^{-x/2}/(2^{(d-1)/2}\Gamma((d-1)/2)).$$

Tämän jakauman avulla voimme määrätä tyypilliset arvot χ^2:lle ja erityisesti kiinnittää raja-arvot, jotka osoittavat että testisuureen arvo on poikkeuksellisen pieni tai suuri. (Säännöllinen tasaisesti jakautunut pisteistö johtaa liian pieneen arvoon, epätasainen jakauma liian suureen.)

Koska testisuure noudattaa yo. jakaumaa vain approksimatiivisesti, menetelmää voi käyttää vain suhteellisen suurilla näytemäärillä. Nyrkkisääntö on, että kunkin luokan osumien odotusarvon tulisi olla yli 5, joten optimitapauksessakin (eri luokat yhtä todennäköiset) havaintojen määrän tulisi olla vähintään $5d$.

Yleensä huonotkin satunnaislukugeneraattorit läpäisevät yllä esitetyn testin. Astetta parempi testi saadaan kun tulkitaan satunnaislukujono koostuvaksi tason tai avaruuden pisteiden koordinaateista. Tason tapauksessa ryhmitellään satunnaisluvut $\{(U_1, U_2), (U_3, U_4), \ldots\}$ yksikköneliön pisteiksi ja jaetaan neliö $d = s^2$ pienempään neliöön. Laskemalla osumien määrä kussakin neliössä ja soveltamalla χ^2 testiä, voidaan löytää mahdollinen riippuvuus kahden peräkkäisen satunnaisluvun välillä. Tämä voi syntyä mm. liian pienen kertovan tekijän käytöstä Lehmer generaattorissa. Vastaavasti voidaan toimia lukukolmikoiden tapauksessa jne. Ongelma on vain se, että dimension kasvaessa tarvittavien luokkien määrä kasvaa nopeasti ja testistä tulee aivan liian kallis.

2.4.2 Peräkkäiskorrelaatio

Edellä mainittu lukuparien ja -kolmikoiden testaus kertoo jotakin peräkkäisten satunnaislukujen korrelaatiosta (jota tietysti ei pitäisi olla lainkaan). Jos tarvitsemme tietoa pidempien sekvenssien (5 tai useampi satunnaislukua) riippuvuuksista edellä mainittu strategia on liian kallis. Tarvitaan siis halvempia menetelmiä. Varsin yksinkertainen testi on muodostaa k luvun joukkoja, joista kustakin etsitään maksimiarvo. Asetetaan siis satunnaismuuttujaksi $X = \max(U_1, U_2, \ldots, U_k)$. Jos U:t ovat riippumattomia ja tasajakautuneita, X:n jakauma tunnetaan. Kertymäfunktio on $F(x) = x^k$.

On siis testattava noudattaako havaittu X jakaumaa F. Tätä voidaan testata edellämainitulla χ^2-testillä. Muitakin vaihtoehtoja on, esimerkiksi ns. Kolmogorov-Smirnov testi. Siinä otetaan n havaintoa (X:stä), jotka järjestetään nousevaan järjestykseen. Tämän jälkeen muodostetaan testisuureet

$$D^+ = \max_{1 \le i \le n} (i/n - F(X_i), \qquad D^- = \max_{1 \le i \le n} (F(X_i) - (i-1)/n).$$

Molemmille testisuureille tunnetaan taulukoidut jakaumat, joiden avulla voidaan päätellä ovatko D arvot tyypillisiä vai onko syytä epäillä, että X ei noudata jakaumaa F.

Toinen peräkkäiskorrelaatiota mittaava testi on välitesti (gap test). Olkoon annettu jono tasajakautuneita satunnaislukuja väliltä $(0, 1)$. Valitaan osaväli $0 \le a < b \le 1$. Nyt etsitään lukupareja U_j, U_{j+k} siten, että $U_j, U_{j+k} \in [a, b]$ ja $u_{j+i} \notin [a, b]$ kaikille $1 \le i < k$. Sanomme, että tässä tapauksessa välin pituus on k. Etsitään nyt lukujonosta eripituiset välit. Teoreettisesti välien pituuksien tulisi noudattaa geometrista jakaumaa $P(k) = \beta^{k-1}(1-\beta)$ missä $\beta = 1 - (b - a)$. Havaittujen välinpituuksien sopivuutta tähän jakaumaan voidaan testata χ^2-testillä.

Lisää testejä löytyy mm. Knuthin kirjasta. Siellä esitellään mm. taajustason testi (spectral test), jolla on voitu eliminoida useita käytännössäkin huonoksi todettuja generaattoreita. Tämä testi perustuu satunnaislukujonon tulkinnalle ajasta riippuvaksi signaaliksi ja signaalin taajuusominaisuuksien analysoinnille. (Täysin satunnaisen signaalin tulisi olla ortogonalinen minkä tahansa taajuuskomponentin kanssa.)

Yhteenvetona todettakoon, että satunnaislukugeneraattorien huolellinen testaus on vaativa ja kallis operaatio. Hyvin pitkälle se voidaan katsoa myös tarpeettomaksi, koska valmiiksi testattuja hyviä algoritmeja on tarjolla. Toisaalta testaaminen on myös erittäin tarpeellista, koska myös huonoja generaattoreita on yleisesti tarjolla, jopa 'luotettavista' lähteistä. Yleisesti ottaen on suositeltavampaa käyttää algoritmia, josta on saatavissa dokumentoidut ominaisuudet kuin algoritmia, josta ei tiedetä mitään (valmistajan/ohjelmiston

oma 'salainen' menetelmä). Lisäksi, jos sovelluksella on jotain erityispiirteitä, tähän piirteeseen painottuvia testejä voi olla syytä tehdä vaikka generaattorilla olisi yleisesti hyvä maine. Esimerkiksi, jos satunnaislukuja tarvitaan systemaattisesti k luvun paketteina (vaikka Monte Carlo integroinnissa), k askeleen peräkkäisominaisuudet voivat olla hyvin kriittisiä.

2.5 Generaattoreiden toteutus

Tyypillisessä simuloinnissa generoidaan paljon satunnaissuureita, joiden avulla tehdään hyvin yksinkertaisia operaatioita. Näin ollen tehokas generaattori on oleellinen koko simulointiohjelmiston tehokkuudelle. Tämä puoltaa ainakin kriittisimpien osien koneenläheistä toteutusta. Toisaalta, jos pyritään tekemään yleiskäyttöisiä ja siirrettäviä ohjelmistoja, koneriippuvia piirteitä ei tulisi käyttää. Minimissään kaikista algoritmeista tulisi olla koneriippumattomat korkean tason kielillä tehdyt toteutukset.

Tämä yleensä rajoittaa jo algoritmien valintaa. Esimerkiksi kaikkia kongruenssigeneraattoreita ei voi toteuttaa korkean tason kielten perusaritmetiikalla ainakaan suoraviivaisesti tuetun lukualueen pienuuden takia. Esimerkiksi yleisimmin käytetyt kongruenssigeneraattorit perustuvat kantaluvun $m = 2^{31} - 1$ käyttöön. Normaali kokonaislukuaritmetiikkaa käyttäen siemenluvun ja kertoimen a tulo karkaa lähes aina lukualueelta. Käytettäessä suhteellisen pieniä kertoimia kuten $a = 16807$ välitulokset voi vielä esittää tarkasti, jos käytetään kaksoistarkkuuden liukulukuja, joissa mantissalle on varattu enemmän kuin 46 bittiä:

```
module lehmer
use prec
real(dp),parameter :: m=2**31-1
real(dp) :: a,m_1,seed

contains

subroutine init_lehmer(iseed)
integer :: iseed
m_1=1._dp/m
a=16807._dp
seed=real(iseed,dp)
return
end subroutine init_lehmer

real(wp) function random()
seed=modulo(seed*a,m)
random=seed*m_1
return
end function random

end module lehmer
```

Koneriippuvalla toteutuksella vastaava algoritmi voidaan mahdollisesti toteuttaa myös suuremmilla kertoimilla, jos esimerkiksi tiedetään, että liukulukuyksikkö käyttää sisäisesti suurempaa tarkkuutta.

Tehokkain toteutus edellyttää yleensä konekielistä toteutusta, jossa otetaan myös huomioon kantaluvun erityinen muoto ('melkein' 2^{31}). Tällöin jakolasku/modulo voidaan korvata yksinkertaisemmilla operaatioilla. Tällöin tarvitaan 'shift' operaatio. Samoin, jos halutaan tarkastella sekoitettua varianttia, jossa osa satunnaisluvun biteistä tulkitaan pieneksi kokonaisluvuksi, pääsy bittitasolle on välttämätön. Myös Marsaglia-MacLaren normaalijakaumageneraattori edellyttää ainakin virtaviivaisimmillaan bittitason informaation käyttöä (ks. Knuth).

Toinen toteutukseen liittyvä aspekti on generaattoreiden ohjelmistorajapintojen määrittely. Koska kaikkien jakaumien generointiin käytetään lopulta $(0, 1)$ tasajakaumaa, on päätettävä käyttävätkö eri generaattorit yhteistä tasajakaumajonoa vai luodaanko kullekin jakaumalle omat jononsa, jolloin jokaiselle tulee määritellä alkusiemenet ja suorittaa tarvittavat alustukset (esim. jos käytetään sekoitettua generaattoria).

Oliopohjaisissa toteutuksissa satunnaislukugeneraattorit toteutetaan yleensä siten, että perusluokassa määritellään alustukset ja rutiinit tasajakautuneille $(0, 1)$ satunnaisluvuille. Tämä luokka periytyy alaluokille, joita on yksi jokaista tarvittavaa jakaumatyyppiä kohden. Jakaumien numeeriset parametrit välitetään parametrilistassa. Java- koodattu esimerkki löytyy mm. JavaSIM ohjelmiston lähdekoodeista, ks. verkkosivut.

Luku 3

Diskreettiaikainen simulointi

Diskreettiaikaisessa, tai tapahtumapohjaisessa, simuloinnissa tarkastellaan systeemejä, jotka koostuvat selkeästi määritellyistä alisysteemeistä, jotka kommunikoivat toistensa kanssa *tapahtumien* välityksellä. Tapahtuma on tiettyyn aikaan sidottu toimenpidesarja, joka muuttaa osasysteemien tiloja ja mahdollisesti generoi muita tapahtumia samalle tai myöhäisemmälle ajalle.

Tyypillisesti tapahtumapohjaista simulointia käytetään erilaisten jonomallien käsittelyyn. Yksinkertaisimmillaan kyse on prosessista, jossa jono asiakkaita kulkee järjestyksessä yhden tai useamman peräkkäisen palvelupisteen läpi ja simuloinnilla halutaan tietoa mahdollisesta jononmuodostuksesta, odotus- ja läpimenoajoista jne. Vastaavantyyppisiä malleja voidaan rakentaa myös varastointi- ja kuljetussysteemeille.

Kaikille diskreettiaikaisille simulointimalleille on löydettävissä viisi yhteistä osatehtävää, jotka simulointiohjelmiston tulee hallita.

Ensimmäisenä tarvitaan kalusto *mallin rakenteen* spesifiointiin. Tämä tarkoittaa käytännössä kolmea asiaa: systeemin osien, osien välisen loogisten riippuvuuksien ja systeemissä esiintyvien tapahtumien mallittamista. Ensimmäisessä kohta sisältää osasysteemien tilojen karakterisoinnin diskreettien muuttujien avulla, toinen osa mallin loogisen verkostorakenteen (vuokaavio) ja kolmas osa kuvaa sisältää varsinaisen toiminnan yleensä funktioina ja proseduureina, jotka muokkaavat osasysteemien tilamuuttujia.

Toinen komponentti on *ajan hallinta*. Diskreettiaikaisissa systeemeissä aika etenee vaihtelevan mittaisin askelin tapahtumahetkestä seuraavaan.

Kolmannen osan-alueen muodostavat *satunnaisprosessit*, joita käytetään mallin datan tuottamiseen (asiakkaiden tuloaikataulut, palvelujen kestot). Tämän osa sisältää yleensä joukon satunnaislukugeneraattoreita, joiden tulisi kattaa kaikki tarvittavat jakaumat.

Simuloinnin tulosten tarkastelua varten tarvitaan *tilastollinen raportointi*, jonka avulla kerätään ja muokataan halutut tiedot simuloinnin tuloksista. Tämän osan tulisi tuottaa tarvittavat tilastolliset tunnusluvut (keskiarvot, varianssit, luottamusvälit) sekä graafiset esitykset (histogrammit, aikasarjat).

Viimeisenä mutta ehkä keskeisimpänä tarvitaan simuloinnin *hallintasysteemi*, joka kontrolloi mallin tapahtumia. Käytännössä tämä tarkoittaa prosessia, joka pitää järjestyksessä tulevia tapahtumia ja on vastuussa siitä, että tapahtumat aktivoituvat oikeassa järjestyksessä. Simulointimallin luonteesta paljolti riippuu se, miten ns. *tapahtumalista* tulisi järjestää. Tarvittavia toimintoja ovat seuraavan tapahtuman etsiminen, uuden tapahtuman lisäys ja tapahtuman poisto. Tarvittavat tietorakenteet ja algoritmit on valittava sen

mukaan mitkä toiminnot ovat kulloinkin dominoivia.

Tarkastelemalla simulointimallia eri näkökulmista päädytään hyvin erilaisiin tapoihin toteuttaa simulointiohjelmisto. Lähtien mallin rakenteen kolmiosaisuudesta päädytään kolmeen mahdolliseen tarkastelukulmaan: tapahtuma-, prosessi- tai aktiviteettilähtöiseen simulointiin. *Tapahtuma* on joukko samaan aikaan sidottuja tilan muutoksia. Sama tapahtuma voi koskea useampaa systeemin osaa (esimerkiksi asiakas, joka poistuu palvelusta X, näkyy välittömästi palvelussa Y joko käynnistyvänä palveluna tai jonon pitenemisenä.) *Prosessi* puolestaan on järjestetty jono tapahtumia, jotka liittyvät yhteen osasysteemiin muodostaen tämän osan *elinkaaren*. Tällöin kaikki mainittua osaa koskevat toiminnot voidaan koota yhteen ja samaan prosessiin. Lopuksi *aktiviteetti* on osasysteemin yksittäinen aikaa vievä toiminto. Aktiviteettiin liitetään joukko ehtoja, joiden toteutuessa se voi käynnistyä. Esimerkiksi pesuaseman pesuaktiviteetti voi käynnistyä jos asema on vapaa ja jonossa on vähintään yksi asiakas. Aktiviteetin aikana asema on varattu, joten samaan asemaan ei voi syntyä uutta aktiviteettia, ennenkuin edellinen on ohi.

Tarkastellaan nyt lyhyesti edellä olevan pohjalta kolmea vaihtoehtoista simulointilogiikkaa.

Tapahtumalähtöisessä simuloinnissa keskeisessä asemassa ovat *tapahtumarutiinit*, joita on yksi kutakin tapahtumatyyppiä kohden. Tapahtumien järjestelijä (scheduler) pitää listaa *tapahtumailmoituksista* (event notice), jotka sisältävät vähintään tapahtuman ajan ja referenssin a.o. tapahtumarutiiniin. Yksittäinen tapahtuma voi generoida tai peruuttaa muita tapahtumia muuttamalla tapahtumailmoitusten listaa. Jokaisen tapahtuman jälkeen kello asetetaan seuraavan tapahtuman aikaan.

Prosessilähtöinen simulointi liittyy läheisesti *olio-pohjaiseen* ohjelmointiin. Kustakin osaprosessista muodostetaan olio, joka sisältää tarvittavat tilamuuttujat sekä rutiinit olion elinkaaren eri tapahtumille. Olio kommunikoi ympäristönsä (scheduler, toiset oliot) kanssa tarkoitukseen varattujen proseduurien välityksellä. Olio voi esimerkiksi lähettää toista oliota koskevan aktivointi- tai passivointimääräyksen.

Aktiviteettipohjaisessa lähestymistavassa malli koodataan aktiviteettirutiineihin, joissa on kussakin kaksi rajapintaa. Aloitusrajapinta tarkistaa ovatko kaikki aktivointiehdot voimassa ja varaa tarvittavat resurssit aktiviteetin ajaksi sekä kiinnittää lopetusajan. Lopetusrajapinnassa tehdään vastavasti lopetukseen liittyvät toimenpiteet ja vapautetaan resurssit. Mallia hallinnoidaan käymällä systemaattisesti läpi kaikki aktiviteetit tarkistaen voidaanko ne aloittaa. Tätä toistetaan kunnes yksikään uusi aktiviteetti ei aktivoidu. Tällöin siirretään kelloa seuraavaan tapahtuma-aikaan, jolloin jokin aktiviteetti päättyy ja eri aktiviteettien aloitusehdot mahdollisesti päivittyvät. Tämä lähetymistapa on laskennallisesti kallis mutta mallin logiikan toteuttamisen suhteen varsin yksinkertainen, koska kullekin toiminnolle on vain määriteltävä aloitus- ja lopetusehdot ja toimenpiteet.

Jatkossa tarkastelemme lähemmin tapahtuma- ja prosessipohjaista lähestymistapaa. Kummassakin tapauksessa käytämme esimerkkinä johdannossa esitettyä yksinkertaista jonomallia eli pesuautomaattiesimerkkiä.

3.1 Tapahtumapohjainen lähestymistapa

Tapahtumapohjaisessa lähestymistavassa simulointimallin logiikka koodataan joukkoon tapahtumarutiineja. Mikäli mahdollisia tapahtumatyyppejä on vähän ja systeemin rakenne on yksinkertainen tämä voidaan tehdä tehokkaasti useimmilla korkeantason kielillä. Eri-

tyisesti ilman kaikkia oliokielten piirteitä. Toisaalta suurille systeemeille mallin logiikan hallinta vaikeutuu kun osasysteemien hallinta fragmentoituu moniin tapahtumarutiineihin.

Tarkastellaan yksinkertaista jonomallia, joka jaetaan neljään eri tapahtumaan. Annetaan näille nimet *Tulo* (generoi jononpäähän tulevan asiakkaan), *Alku* (käynnistää pesun), *Loppu* (lopettaa pesun) ja *Lähtö* (poistaa asiakkaan systeemistä).

Nyt mallin logiikka tapahtumista käsin on pääpiirteittäin seuraava: Tulo-tapahtuma tarkistaa onko jonossa tilaa, jos ei todetaan, että asiakas meni ohi, muuten luodaan asiakas ja lisätään jonoon ja lisätään tapahtumalistaan p.o. asiakkaan Alku-tapahtuma samalle ajanhetkelle. Tämän jälkeen määrätään seuraava Tulo-aika sopivasta jakaumasta ja lisätään Tulo-tapahtuma tapahtumalistaan.

Alku-tapahtuma tarkistaa onko asema vapaa ja jonossa asiakkaita. Jos ei ole, ei tehdä mitään. Jos asema on vapaa ja jonossa asiakas, otetaan jonosta ensimmäinen asiakas, varataan asema, määrätään pesun kesto sopivasta jakaumasta ja lisätään tapahtumalistaan Loppu a.o. ajalle.

Loppu-tapahtumassa vapautetaan asema, lisätään Lähtö tapahtumalistaan (samalle ajalle). Samoin lisätään uusi Alku-tapahtuma samalle ajalle.

Lähtö-tapahtumassa kerätään asiakkaasta tarvittavat tilastotiedot (esim. jos halutaaan tutkia läpimeno tai jonotusaikoja) ja poistetaan asiakas.

Edellinen tapahtumajako ei ole ainut mahdollinen. Esimerkiksi Lähtö voitaisiin integroida Loppu-tapahtumaan. Samoin voitaisiin muodostaa tapahtumat Tulo/Alku ja Loppu/Alku sensijaan että generoidaan välittömästi laukeavia tapahtumia. Tällöin tosin Alku-tapahtuma kopioituu kahteen eri tapahtumaan, jolloin on huolehdittava siitä, että molemmat versiot toimivat identtisesti. Esimerkiksi, jos palvelun kestoaikajakaumaa muutetaan, se on tehtävä molempiin kopioihin. Tämä antanee jo kuvan siitä, kuinka mallin logiikka fragmentoituu ja muuttuu nopeasti vaikeasti hallittavaksi, systeemi kasvaa.

Ylläolevat toiminnot voidaan koodata erillisiksi tapahtumarutiineiksi. Tämä edellyttää, että tarvittaville suureille on olemassa sopivat tietorakenteet ja niitä tukevat rutiinit. Tarkastellaan aluksi tapahtumien käsittelyyn liittyviä rakenteita. Jotta voimme viitata tiettyyn tapahtumaan, tarvitsemme muuttujia, jotka voivat saada arvokseen tapahtuman tyypin. Tätä varten on luonnollista määritellä tyyppi TapahtumaTyyppi, jonka mahdollisia arvoja ovat *Tulo, Alku, Loppu, Lähtö*.

Kustakin tapahtumasta tehdään etukäteen tapahtumailmoitus, joka on muuttujatyyppiä IlmoitusTyyppi ja sisältää kentät *Aika, Tapahtuma, SeuraavaTapahtuma*. Näistä *Aika* on reaaliluku, *Tapahtuma* tyyppiä TapahtumaTyyppi ja *SeuraavaTapahtuma* osoitin seuraavaan tapahtumailmoitukseen.

Tapahtumailmoituksista pidetään tapahtumalistaa, joka voi olla esimerkiksi yksinkertainen linkitty lista, jossa tapahtumat ovat aikajärjestyksessä. Listan käsittelyyn on määriteltävä ainakin rutiinit **LisääTapahtuma**(*Tapahtuma, Aika*), joka tekee tapahtumasta ilmoituksen ja sijoittaa sen oikealla paikalle listaan, sekä **SeuraavaTapahtuma**, joka ottaa listasta ensimmäisen tapahtumailmoituksen käsiteltäväksi. Lisäksi tarvitaan **PoistaTapahtuma**(*Tapahtuma*), joka poistaa tarpeettomaksi käyneen tulevaa tapahtumaa vastaavan ilmoituksen (etsii ilmoituksen listasta, poistaa sen listasta ja vapauttaa muistin). Tarpeen mukaan listasta haluta myös muuta tietoa kuten sen pituus tai tieto siitä onko lista tyhjä. Debuggausta varten on syytä olla myös rutiini, joka tulostaa koko listan haluttaessa.

Varsinaista simulointia hallinnoi monitorirutiini, joka ottaa tapahtumalistasta kulloinkin ensimmäisen tapahtuman, selvittää sen tyypin ja kutsuu vastaavaa tapahtumarutiinia. Tämän jälkeen poistetaan käsitelty tapahtumailmoitus ja tarkastetaan onko simulointi syytä päättää. Ts. onko simuloitu riittävän pitkä aika, riittävän monta asiakasta tai onko systeemi tyhjentynyt (jos esimerkiksi on valittu toteutustapa, jossa Tulo rutiini generoi vain äärellisen määrän uusia asiakkaita). Riippuen valitusta tavasta määrätä simuloinnin kesto, monitori tarvitsee erilaisia tietoja kellonajasta, jonon pituudesta tai simuloitujen asiakkaiden/tapahtumien määrästä. Näille suureille on määriteltävä tarvittavat funktiot ja laskurit sekä alustettava ne ennen simulointia.

Toinen aspekti, joka on kiinnitettävä tapahtumien hallinnan lisäksi, on asiakkaiden käsittely. Periaatteessa esimerkin tehtävänannon kannalta asiakkaista ei haluta mitään erityistä tietoa, joten asiakkaille ei tarvitse asettaa mitään erityisominaisuuksia. Minimissään tässä esimerkissä ei asiakkaita tavallaan tarvita lainkaan vaan riittää, että hallitaan palvelun (eli Alku-tapahtuman) eteen kertyvä jono. Jos kuitenkin halutaan seurata esimerkiksi asiakkaiden läpimeno tai odotusaikoja, kustakin asiakkaasta on luotava oma muuttujansa. Tämä voi olla esimerkiksi tyyppiä AsiakasTyyppi, jossa on kaksi kenttää, *TuloAika* ja *SeuraavaAsiakas*, joka on osoitin seuraavaan AsiakasTyyppi-muuttujaan. Osoitinmuuttujaa tarvitaan, jos jonot käsitellään linkittyinä listoina. Esimerkissä tosin jonon maksimipituus tunnetaan, jolloin staattinen tietorakenne (taulukko) on myös mahdollinen.

Jonon hallintaan tarvitaan tavanomaiset rutiinit, jotka lisäävät asiakkaan jonoon, poistavat ensimmäisen asiakkaan, ilmoittavat onko jono tyhjä tai täysi.

Nyt tapahtumarutiinit voi hahmotella seuraavasti:

```
Tulo_Tapahtuma()
  Asiakas_Tyyppi_Osoitin :: Auto
  {
  Lisää_Tapahtuma(Tulo_Aika_Jakauma(),Tulo)
  Auto= Luo_Asiakas()
  Jono.Lisää(Auto)
  Lisää_Tapahtuma(0.,Alku)
}

Alku_Tapahtuma()
  Asiakas_Tyyppi_Osoitin :: Auto
  {
  If(Asema.Vapaa() and Jono.Pituus()>0) then
     Auto=Jono.Ota()
     Asema.Varaa(Auto)
     Lisää_Tapahtuma(Palvelu_Aika_Jakauma(),Loppu)
  Endif
}

Loppu_Tapahtuma()
  Asiakas_Tyyppi_Osoitin :: Auto
  {
  Auto= Asema.Vapauta()
  Lähtö.Varaa(Auto)
```

```
  Lisää_Tapahtuma(0.,Lähtö)
  Lisää_Tapahtuma(0.,Alku)
}

Lähtö_Tapahtuma()
  Asiakas_Tyyppi_Osoitin :: Auto
  {
   Auto=Lähtö.Vapauta()
  // Kerää statistiikkaa
  Poista_Asiakas(Auto)
}
```

Käytännössä kannattaa varautua malleihin, joissa voi olla useita palvelimia, jonoja, asiakastyyppejä jne. Tällöin vastaavat apurutiinit tulisi määritellä niin, että jonot ym. voidaan identifioida ja niitä vastaavat muuttujat kulkevat parametrilistoissa. Määritellään siis eksplisiittisesti mitä jonoa, palvelinta, asiakastyyppiä jne kulloinkin käsitellään.

Edellä tietyille tapahtumille täytyi välittää eksplisiittisesti, mitä asiakasta tapahtuma koskee, koska ko. tapahtumilla ei ollut omia jonoja. Tätä varten on on oletettu globaalit osoitinmuuttujat, joihin viitataan Varaa ja Vapauta metodeissa. Ts samalla kun resurssi varataan asiakkaan käyttöön, asiakkaan osoitin talletetaan, jotta resurssin vapauttava tapahtuma löytää oikean asiakkaan.

Ylläolevissa tapahtumarutiineissa jätettiin huomiotta kaikki tilastotiedon keruu, joka simuloinnin tulosten kannalta on kuitenkin olennaisin. Tarpeista riippuen tämä voidaan tehdä useammalla tavalla liittäen informaatio joko asiakkaisiin, palvelimiin tai globaaleihin laskureihin, joita operoidaan joko erikseen tai eri operaatioiden yhteydessä. Yleispätevän tilastointisysteemin luominen on vaativa operaatio ja helposti ajaudutaan vaihtoehtoon, jossa kysymyksenasettelun vaihtuminen pakottaa käymään läpi ja modifioimaan tapahtumarutiinit, jotta halutut suureet saadaan laskettua. Jos erilaisia tapahtumia ja vastaavia rutiineja on paljon, tämä ylläpitotyö vaikeutuu helposti.

3.2 Prosessi- tai oliopohjainen simulointi

Tapahtumapohjaisessa simuloinnissa mallin logiikka fragmentoituu helposti lukuisiin irrallisiin tapahtumarutiineihin. Prosessipohjaisessa tarkastelussa loogisesti yhteenkuuluvat tapahtumat muodostavat prosessin, jolla on tietty oma elinkaarensa. Elinkaaren muodostaa jono peräkkäisiä tapahtumia, jotka voidaan koota samaan yksikköön (ohjelmalohkoon).

Esimerkissämme luonteva prosessi on asiakkaan historia systeemissä. Koska asiakkaita on vain yhden tyyppisiä, tarvitaan minimissään vain yksi prosessi ja sitä vastaava elinkaari, joista generoidaan tarvittava määrä kopioita. Periaatteessa prosesin elinkaari on varsin yksinkertainen: Asiakas generoidaan ja samalla allokoidaan seuraava asiakas generoitavaksi määrätyn ajan kuluttua. Asiakas asetetaan palvelun jonoon. Ajallaan asiakas otetaan jonosta ja varataan palvelu tietyksi ajaksi, jonka jälkeen palvelu vapautetaan ja asiakas siirtyy pois systeemistä.

Käytännön toteutus ei ole aivan näin suoraviivainen. Mikäli kirjoitetaan yksi ainoa aliohjelma, joka kuvaa ym. elinkaaren, miten huomioidaan se, että prosessissa on samanaikaisesti useita, eri vaiheissa olevia asiakkaita. Ts. tavallaan useita rinnakkain suoritettavia

versioita samasta aliohjelmasta. Asia voidaan ratkaista periaatteessa varaamalla kutakin asiakasta kohden prosessilaskuri, joka kertoo missä kohti prosessia a.o. asiakas on. Esimerkissämme asiakas käy neljän tapahtuman läpi, joten aliohjelma voidaan jakaa neljään eri vaiheeseen, joilla on tunnukset. Tällöin tilanne voidaan hallita, jos tapahtumalistassa on viittaus tulevien tapahtumien aikaan, prosessiin, joka aktivoituu ja prosessin senhetkiseen vaiheeseen.

Perinteisillä korkeantason kielillä prosessiorientoitunut simulointi on toteutettava siten, että prosessia kuvaava aliohjelma palauttaa kontrollin kutsuvaan aliohjelmaan jokaisen vaiheen jälkeen ja kääntäen, kutsuttaessa siirtyy suoraan kulloinkin tarvittavaan vaiheeseen. Aliohjelman rakenne on hahmoteltu alla.

```
Asiakasprosessi(Vaihe)
  VaiheTyyppi :: Vaihe
  {
  CASE Vaihe

Tulo {
  UusiProsessi(TuloAikaJakauma(),Tulo)
  Jos (JononPituus< m){
    Auto =new Asiakas()
    Jono.Lisää(Auto)
    KysyPalvelijaa()
    }
  }
Alku {
  SeuraavaVaihe(PalveluAikaJakauma())
  }
Loppu {
  VapautaPalvelija()
  KysyPalvelijaa()
  SeuraavaVaihe(0.)
  }
Lahto {
  //Keraa statistiikka
  PoistaAsiakas
  }
  ENDCASE
}
```

Tätä runkoa on vielä täydennettävä prosessiriippuvalla jononkäsittelyllä. Nimittäin rutiini KysyPalvelijaa(), huolehtii asiakkaiden siirtymisestä Alku-vaiheeseen. Tämä tapahtuu seuraavasti:

```
KysyPalvelijaa()
AsiakasTyyppi :: Auto
{
```

```
 Jos(PalvelijaVapaa() ja Jono.Pituus()>0) {
   Auto=Jono.Ota()
   VaraaPalvelija()
   SeuraavaVaihe(0.,Auto)
   }
}
```

Tämä rutiini aktivoituu aina kun jonon pituus tai palvelimen työtilanne muuttuvat. Huomattakoon, että rutiini siirtää seuraavaan vaiheeseen yleensä asiakkaan, joka on eri kuin kutsuva prosessi. Siispä yleensä on varauduttava välittämään viittaus asiakkaaseen parametrilistassa. Sama pätee tietysti palvelimiin, jonoihin jne, joita yleisessä mallissa voi olla useita. Ylläolevasta nämä viittaukset on lyhyyden takia jätetty pois.

Ylläolevassa lähestymistavassa kustakin prosessista talletettiin muistiin asiakkaan vaatimat tiedot, sekä prosessin vaihe. Tämä on käytännössä vähän hankala ja työläs tapa. Esimerkiksi, jos halutaan että prosessi muistaa jotain parametrejaan, ne on koodattava osaksi asiakkaan tietorakennetta. Samoin prosessin vaiheiden kirjanpito on työlästä rutiinia. Helppokäyttöisempi systeemi saadaan, jos voidaan olettaa, että prosessi muistaa omat parametrinsa kutsujen välissä, samoin kuin paikan, josta prosessista on poistuttu/johon tulee palata. Tämä voidaan toteuttaa muodostamalla prosesseista oliot, joissa prosessin tila on koodattu olion sisäisiin muuttujiin, samoin kuin prosessin vaihe. Lisäksi on määriteltävä olion metodit siten, että ne tarvittaessa tallettavat automaattisesti informaation siitä, miisä kohden prosessia suoritus on, samoin kuin missä tilassa prosessi on.

Vakiintunut tapa oliopohjaisessa simuloinnissa on määritellä prosesseille neljä mahdollista tilaa: aktiivinen (eli parhaillaan suoritettava prosessi), ajastettu (scheduled, prosessi, jonka aktivoitumiselle on määrätty aika), passivinen (prosessi, joka ei aktivoidu ilman eri toimenpiteitä) ja lopetettu (prosessi, jota ei voi enää aktivoida).

Prosesseihin (olioihin) liittyy muutamia menetelmiä, joilla niiden tilaan voi vaikuttaa. *Activate()* rutiinia sovelletaan yleensä passiiviseen olioon, joka aktivoidaan joko välittömästi tai jonkin viiveen, kellonajan tai tapahtuman jälkeen. Tällöin kohde oliosta tulee siis joko aktiivinen tai ajastettu. Voidaan sopia, että käsiteltäessä jo ajastettuja olioita metodin nimi on eri, esim. *ReActivate()*. Aktiivinen voidaan passivoida metodilla *Passivate* ja ajastettu metodilla *Cancel()*. Lisäksi prosessi voi keskeyttää ja ajastaa itsensä metodilla *Hold()*, jolla määrätään aika, jolloin prosessi aktivoituu uudelleen tai *Wait()* johon liitetään viite prosessista, jonka tulee aktivoitua ennenkuin p.o. prosessi tulee aktivoitumaan. Jos prosessi suorittaa kaikki toimintonsa, se lopettaa lopuksi itsensä eli siirtyy tilaan, josta sitä ei voida enää aktivoida (tuhoaa itsensä).

Edellä oleva prosessin tilojen luokittelu ja niitä koskevien menetelmien määrittely on peräisin SIMULA-kielestä, joka kehitettiin 60-luvulla diskreettiaikasimulointiin. SIMULA on kaikkien olio-kielien esikuva, joka ensimmäisenä toteutti monet olio-ohjelmoinnin keskeiset piirteet kuten luokat, periytymisen ja oliot.

Esimerkkimme voidaan mallittaa oliopohjaisesti monella eri tavalla riippuen siitä mihin toimijoihin aktiviteetit sijoitetaan. Esimerkiksi onko asiakas aktiivinen olio, joka varaa passiivisen aseman, vai onko asiakas passiivista massaa, jota aktiiviset palvelijat käsittelevät.

Tarkastellaan asiakas-pohjaista toteutusta. Olkoon Q passiivinen asema, jolla on omat jonotus- ja varausmetodit. Tällöin asiakkaan rutiini voi olla muotoa:

```
\\Asiakasprosessi
Asiakas Auto
Asema Q
Auto = new Asiakas
Auto.Activate(TuloAikaJakauma()) \\ajastetaan seuraava tulo
Jos (Q.Pituus <m+1)
  {Jos(Q.Varattu)
   {Q.Lisaa(*this) \\lisataan nykyinen asiakas jonoon
    Passivate      \\ kontrolli siirtyy pois nykyisesta prosessista
   }                \\ ja palaa vasta kun asiakas aktivoidaan
   Q.VaraaPalvelin \\ varataan palvelu mahd. jonotuksen jalkeen
   Hold(PalveluAikaJakauma())
   Q.VapautaPalvelin
   Jos(Q.Pituus >0)
    {Asiakas Auto2 = Q.Ota
     Auto2.Activate(0.)
    }
   \\ Kerataan statiikka
   }
Terminate \\tuhotaan asiakas (delete)
}
```

Simulointiprosessi itsessään voidaan myös toteutta oliona, jolla on vastaavat metodit kuin osaprosesseilla. Tällöin scheduler huolehtii tarvittavista kontrollin siirroista osaprosessien ja pääprosessin välillä. Esimerkin simulointi voi sujua esimerkiksi seuraavalla tavalla

```
AlustaStatiikka
Q = New Jono
Auto = New Asiakas
Auto.Activate(TuloAikaJakauma())
Hold(SimuloinninKesto)
Raportoi
Terminate
```

Kontrolli siiretään siis asiakas-prosessille, joka aikanaan aktivoi seuraavan asiakkaan, liittyy mahdollisesti jonoon, varaa ja vapauttaa aseman resurssit. Kun simulointi on edennyt määrätyn ajan, kontrolli palaa kutsuvaan ohjelmaan, jossa voidaan tehdä tarvittavat jatkotoimet.

Käytännössä esimerkin toteutus ei ole aivan niin suoraviivaista, kun edellä esitetään. Uudet asiakkaat generoidaan tässä edellisten (järjestelmästä poistuvien) olioiden sisällä, mikä on omiaan aiheuttamaan hämminkiä ohjelman suorituksen aikana.

Edelläoleva hahmotelma perustuu mekanismille, joka mahdollistaa aliohjelmaan palaamisen oikeaan paikkaan. Ts. siihen käskyyn, johon suoritus kyseisen olion kohdalta jäi. Tällä tekniikalla eri prosessit voidaan toteuttaa rinnakkaisina ja keskenään tasa-arvoisina korutiineina, joiden avulla usean rinnakkaisen prosessin hallinta on käsitteellisesti helppoa. Sen sijaan korutiinivalmiuden rakentaminen simulointiympäristöön on varsin tekninen toimenpide, joka pitää mielellään piilottaa käyttäjältä mahdollisimman hyvin. Tähän

olio-pohjaisissa simulointijärjestelmissä käytetään periytymistä. Simulointiprosessit määritellään käyttäen pohjana abstrakteja prosessiluokkia, jotka sisältävät valmiiksi korutiinien vaativat toiminnot (kuten Activate, Hold, jne). Näihin lisätään tarvittava sisäinen logiikka ja mahdolliset muut tarvittavat toiminnot (alustukset, tilastoinnit).

Tarkastellaan lopuksi astetta realistisempaa simulointimallia eli yleistä jonoverkostoa. Se koostuu joukosta palvelinsolmuja, joilla kullakin on oma jononsa, sekä yhteyksistä solmujen välillä, jotka kuvaavat töiden mahdollisia reittejä palvelimelta toiselle.

Jonoverkoston spesifioimiseksi on määrättävä seuraavat asiat: solmujen määrä verkostossa ja kussakin solmussa olevan serverin laatu. Kunkin solmun jonotusperiaate (jono, pino, prioriteettijono, tms.), mahdollinen maksimaalinen jononpituus ja tapa miten mahdollinen ylivuoto hoidetaan (esimerkiksi pysähtyykö edellinen palvelin, jos työ ei mahdu seuraavaan jonoon) sekä asiakasprosessien ominaisuudet (asiakkaiden generointi eri solmuihin, todennäköisyydet siirtyä tietystä solmusta toisen solmun jonoon, palveluaikojen jakaumat eri palvelimilla).

Tarkastellaan seuraavaa tapausta. Palvelimia on N kappaletta, kukin omassa solmussaan. Asiakasprosessit ovat tilastollisesti identtisiä (samat palveluaika- yms. jakaumat kaikilla), jonot ovat FIFO-tyyppisiä eikä niiden pituutta ole rajoitettu. Asiakkaat generoidaan yhdessä prosessissa eksponenttijakautunein väliajoin. Uusi asiakas siirtyy palvelimen i jonoon todennäköisyydellä $q_{0,i}$ (tai poistuu välittömästi todennäköisyydellä $q_{0,0}$. Vastaavasti palvelimelta i lähtevä asiakas menee jonoon j todennäköisyydellä $q_{i,j}$ tai poistuu tn:llä $q_{i,0}$. Luonnollisesti pätee $\sum_{j=0}^{N} q_{i,j} = 1$ kaikille $j = 0, \ldots, N$. Kullakin palvelimella on oma palveluaikajakaumansa.

Seuraavassa rakennamme verkolle mallin, joka on yleistettävissä tapaukseen,jossa kussakin solmussa voi olla useampia rinnakkaisia palvelimia. Tätä varten erotamme verkon solmut ja palvelimet toisistaan. Solmuja varten muodostetaan olio Solmu, joka sisältää solmuun liittyvän jonon (Jono), referenssin solmussa olevaan palvelimeen (tai palvelimiin) ja palvelimen ominaisuuksiin. Lisäksi tarvitaan vektori Q, joka sisältää siirtymistodennäköisyydet eri solmuihin. Verkko muodostuu nyt N alkion vektorista, jonka elementit ovat Solmu-tyyppisiä. Solmut eivät ole prosesseja, joten niillä ei ole elinkaarta vaan pelkästään initialisointiin liittyviä apurutiineja, kuten vektorin Q ja palveluaikajakaumien alustus, jonon alustus sekä palvelinprosessin aktivointi.

Itse toiminta voidaan mallittaa kolmen prosessin avulla. Nämä ovat Tulo (asiakkaan luonti), Asiakas ja Palvelin. Tulo-prosessi sisältää perustapaukseen verrattuna myös vektorin Q siirtotodennäköisyyksille, joka tulee alustaa ennen simulointia. Itse prosessin elinkaari on kuten yhden palvelimen tapauksessa, paitsi että kullekin luodulle asiakkaalle arvotaan jono, johon asiakas liittyy. Vastaavasti Palvelin-prosessin on tiedettävä, missä solmussa se sijaitsee. Tämän tiedon avulla Palvelin osaa hakea Asiakkaat oikeasta jonosta, tietää oman palveluaikajakaumansa sekä osaa arpoa Asiakkaalle seuraavan solmun.

Asiakas voidaan mallittaa passiivisena oliona (ilman elinkaarta), jota em. prosessit käsittelevät. Sisäisten muuttujien (tarvittavat tulo- ja muut ajat, laskurit) lisäksi asiakas tarvitsee rutiinin, jolla se siirtyy solmusta toiseen. Ts. kun prosessi siirtää Asiakkaan johonkin jonoon, se kutsuu Asiakkaan rutiinia (esim. Move), jolle annetaan parametrina kohdesolmu, tai vaihtoehtoisesti annetaan kohdesolmujen todennäköisyydet määräävä vektori, jonka avulla Move arpoo itse kohdesolmun ja siirtää asiakkaan sinne. Jonojen käsittelyä varten Asiakkalla on oltava linkitysmekanismit.