Pszeudo-random tesztmanőver generálás robusztussági tesztekhez

Mikes Marcell

# Robusztussági teszt

## Robusztussági tesztek jelenleg

Jelenleg a thyssenkrupp által használt robusztussági tesztek, egy játékszimulációban levezetett mérést játszanak le újra a tesztpadon. A robusztussági teszt futtatása közben pedig hibákat injektálhat a tesztmérnök. A különböző vezetési módokhoz több mérés is van, de összesen limitált mennyiségű, így a tesztesetek száma is limitált. Tesztelés szempontjából jobb lenne, ha bármennyi egymástól eltérő tesztmanővert lehetne futtatni, ezáltal növelve a tesztelt munkapontokat. Több munkapont tesztelése esetén pedig akár olyan hibát is találhatunk a rendszerben, amit a jelenlegi robusztussági teszttel nem vettünk volna észre.

### Tesztelés fontossága

A fejlesztési folyamat fontos része a tesztelés, mert így győződhetünk meg arról, hogy egy átadott termék úgy működik, ahogy azt a megrendelő szeretné. A szoftverfejlesztés során bonyolult rendszereknél körültekintés ellenére akaratlanul is kerülhet hiba a szoftverbe. A cél, hogy ezeket a hibákat még időben beazonosítsuk és kijavítsuk. A thyssenkrupp által fejlesztett termékek biztonságkritikus rendszerek, így kiemelt figyelmet kell fordítani a működésük helyességének biztosítására. A termékeknek továbbá meg kell felelniük az autóipari biztonsági szabványoknak is. Biztonság szempontjából az (ISO26262) és a minőség szempontjából pedig az ASPICE.

# Testpad, tesztkörnyezet

## Elektromechanikus szervokormány

### Kormányrendszer részei

## Testpad

### Testpad részei

# Perlin-zaj

Olyan tesztmanőverhez, ami nem autós mérések alapján készül szükség van egy zaj algoritmusra. Fontos, hogy ne legyen minden lehetséges értéknek ugyan az a valószínűsége, mert akkor a végeredmény fehér zajt lesz, amiből nem lehet járműsebességet vagy kormányszöget készíteni. Ilyen algoritmusokat elsősorban textúrák készítéséhez használnak, mert természetszerű eredményeket lehet velük elérni. Ezek közül én a „Perlin-noise” algoritmust választottam az egyszerűsége és az eredményeinek reprodukálhatósága miatt.

## Bevezető

A Perlin-zajt 1983-ban Ken Perlin… (még nem tudom kell-e)

El lehet készíteni több dimenzióban is (például 3D, 4D), de erre a felhasználásra elég lesz a két dimenzió. Így az algoritmus kettő dimenziós felépítését fogom bemutatni.

## Működése

Képzeljünk el egy tetszőleges hosszúságú négyzetrácsot, ahol a négyzetek mindegyik csúcsához tartozik egy koordináta. Ezen a rácson végig haladva balról jobbra az x, míg fentről lefelé az y koordináta értéke növekszik. Minden ponthoz, azaz csúcshoz rendelni kell egy véletlenszerű gradiens vektort. A kétdimenziós megvalósításban tehát minden négyzethez négy darab vektor fog tartozni. Ezeket a vektorokat aztán egységvektorokká kell alakítani. A vektorok iránya fogja meghatározni az elkészült zajt. Ezután végig menve a négyzetrácson, minden ponthoz négy távolságvektort is definiálunk. A távolságvektorok az adott pont és ahhoz tartozó négyzet csúcsainak a távolságát írják le. Aztán a távolságvektorok és gradiensvektorok skaláris szorzatából ki tudunk számolni négy értéket.

\*kép a rácsról illetve berajzolt gradiens és távolságvektorokról (feltüntetve az aktuális koordinátát, az x y irányt)\*

Ezután a négy kapott értékből bilineáris interpolációval (távolságsúlyozott átlaguk alapján) számoltam ki a jelenlegi pont értékét.

## Tetszőleges méretezés

A [3.2 fejezetben] bemutattam a Perlin algoritmus működését. Látható, hogy az algoritmus egy koordinátapárhoz egy értéket rendel. Ha minden csúcsnak a négyzetrácsban kiszámoljuk az értékét, megjelenítve egy olyan kétdimenziós tömböt kapunk, aminek minden eleme nulla. Ez azért van, mert minden koordináta pontosan a hozzá tartozó négyzet csúcsánál van. Ezért nulla lesz a távolságvektorra és így a skaláris szorzata is.

### Lépésköz

Eddig egy egység oldal hosszúságú négyzetekről volt szó. De ha egy négyzetet több egységre osztunk fel, tehát minden oldala például 100 egység hosszú lesz, akkor a zaj csak minden századik eleme lesz nulla értékű. Ez azt is jelenti, hogy bizonyos időközönként mindig nulla értékes fog felvenni a zaj. A koordinátákat tekintve pedig például a százas egységhossznál maradva, a (101, 50) már a második négyzethez tartozó első és ötvenedik elem lesz. Minél nagyobb egységet rendelünk egy négyzethez, egymáshoz annál közelebbi értékeket kapunk. Ebből következik, hogy az egy egységhez minél jobban közelítünk annál durvább, fehér zajhoz jobban hasonlító eredményt kapunk. A következőkben lépésköznek fogom nevezni a négyzetekhez rendelt egységhosszt.

\*kép (1d) különböző lépésszámú zajokról, esetleg bejelezve, hogy száz lépésszámnál, hogy nézne ki\*

## Átmenetek elsimítása

A Perlin-zaj így már majdnem kész, csak egy dolog hiányzik. Jelenleg a négyzetek között ugrás van és nem átmenet. Ezt a következő „fade” függvénnyel lehet orvosolni, amit Ken Perlin az 2002-es „Improving Noise” tanulmányában írt meg [hivatkozás]

Interpoláció előtt a bal felső távolságvektor koordinátáit kell beilleszteni ebbe a képletbe és aztán az eredményükkel végrehajtani a bilineáris interpolációt. Ahogy az alábbi képen is látható, használata előtt a négyzetrács négyzetei között nem igazán volt átmenet, utána viszont már nem láthatók a négyzetek határvonalai.

\*előtte, utána kép 2d\*

## Oktávok

Az oktáv a zaj részletességét határozza meg. A Perlin-zaj algoritmus esetében ez több különböző frekvenciájú zaj összeadását jelenti. Leggyakoribb megoldás, hogy minden oktávnál a lépésközt és amplitúdót a felére csökkentjük, ezáltal növelve a zaj frekvenciáját. Például egy 100 lépésszámú zajnál két oktáv esetén a 50, háromnál pedig 25 lesz a három különböző zaj lépésszáma. Hasonlóan az amplitúdó is 1-ről 0.5-re majd 0.25-re csökken. A három különböző zajt ezután elég összeadni a három oktáv eléréséhez. Az oktávok segítségével el tudunk érni egy sokkal részletesebb zajt. Tesztmanőver esetében pedig élethűbb kormány- és járműsebesség referenciát kaphatunk vele.

\*vsp több oktávos Perlin-noise kép (esetleg side by side 1 oktávossal, lehet az túl nagy lenne)\*

## Algoritmus igazítása a feladathoz

Annak érdekében, hogy olyan zajt kapjunk, melyből aztán kormányszöget és járműsebességet tudunk csinálni megfelelően fel kell paraméterezni az algoritmust. A zaj által generált elemek száma, frekvenciája és az algoritmus futási ideje is fog számítani tesztmanőver készítésnél. Továbbá fontos az eredmények reprodukálhatósága is.

### Gradiens vektorok generálása

A gradiens vektorokat pszeudo-random generálással fogom létrehozni, a reprodukálhatóság érdekében. A pszeudo-random szám generálás lényege, hogy statisztikailag véletlenszerűnek látszanak a kimenetei, de közben determinisztikus és megismételhető a működése. Az ilyen algoritmusoknak egy kezdőállapotot kell megadni, amit „seed-nek” nevezünk. Azonos kezdőállapotra sorrendben és értékben is megegyező véletlen számokat fogunk kapni. Tesztelésnél fontos, hogy a teszt procedúra reprodukálható legyen. Például egy hiba előfordulásánál meg kell nézni, hogy reprodukálható-e, illetve ez a kiváltó ok megértésénél is fontos lehet. Továbbá a hiba kijavításánál is érdemes ellenőrizni, hogy fennáll-e még a probléma.

A Perlin-zaj algoritmus azonos gradiens vektorokra ugyanazt az eredményt adja. A tesztmanőver elkészítéséhez tehát egy ilyen véletlenszám generátort, a „numpy” python könyvtárból fogok használni. Ezzel generálva a gradiens vektorokat elég csak egy számot, a „seed-et” elmenteni az összes gradiens vektor helyett. Minden elkészült tesztesethez használt „seed” a tesztmanőverekkel együtt tárolva lesz, későbbi újra futtathatóság érdekében.

A feladathoz, én egy 512x512 hosszú négyzetrácsot és ahhoz tartozó gradiens vektorokat generáltam. Gyakorlatban ez két 512 széles és 512 hosszú tömböt jelent, ahol az egyik tömb a gradiens vektorok x míg a másik az y koordinátájukat reprezentálja. Ezt a két tömböt aztán feltöltöttem véletlenszerű számokkal -1 és 1 között. Bármilyen sok gradiens vektort lehet készíteni. Viszont arra kell figyelni, hogy ha a lépésköz túl kicsi és hosszú zajt szeretnénk kapni akkor előfordulhat, hogy több négyzetet akarunk bejárni, mint amennyit csináltunk. Ez túlindexelési hibához vezet, amit megoldhatunk a gradiens vektorok számának növelésével vagy maradékos osztás használatával a hossznál nagyobb indexekre. Utóbbi megoldásnál amint túllépjük a maximum hosszúságot a generált zaj is újra fog kezdődni, mert ugyanazokra a koordinátákra ugyanazokkal a gradiens vektorokkal lesz legenerálva. Azért erre esett a választásom, mert a későbbi futtatások alapján ez volt a nagyság, ahol még nem kellett ismétlődő zajt használnom.

### Lépésköz meghatározása

Tesztmanőver készítéshez fontos a lépésköz helyes megválasztása, mert nagy hatással van a zaj egymás után következő elemeinek távolságára. Ezáltal meghatározza például kormányszög esetében, hogy milyen gyors lesz a kormányszög elfordulásának a sebessége. Ezért a lépésszámot az alapján kell meghatározni, hogy milyen gyors tekerést szeretnénk.

Minden tesztmanőverhez társítok egy időtengelyt is, amivel jelzem a futtatókörnyezet számára, hogy mennyi idő alatt csinálja meg a tesztmanővert. Az időtengely segítségével egy túl nagy frekvenciájú zajt is el lehet nyújtani. A teszt hossza és a lépésszám nagysága egyenesen aránylik egymáshoz. Ugyanaz a frekvencia elérése érdekében kétszer hosszabb tesztnél a lépésszámot is a kétszeresére kell növelni.

### Hosszúság meghatározása

A zaj hosszúságát az határozza meg, hogy hány darab négyzetet járunk be és azok milyen hosszúak. Mivel a gradiens vektorok számát és a lépésszámot is tetszőlegesen állíthatjuk, ezért bármilyen hosszú zajt készíthetünk. Természetesen ahogy az értékek száma, úgy a számítási kapacitás is négyzetesen növekszik.

Fontos megkülönböztetni ezt vagyis a generált zaj elemeinek számát, a teszt hosszúságától. Ideális hosszúságot itt a lépésszám és a gradiens vektorok száma alapján kell beállítani, figyelve a korábban említett dolgokra.

### Oktávok meghatározása

Az oktáv a negyedik olyan változó, amivel a kimenet frekvenciáját módosítani lehet. Az oktávokról elmondható, hogy körülbelül kétharmaddal növelik a frekvenciát. Mivel például két oktávnál két különböző amplitúdójú és lépésszámú zajt kell legenerálni, ezért a számítási kapacitás nő. Minden egyes oktáv a kétszeresére emeli az időt, ami alatt kész lesz a zaj. Ezért érdemes lehet megfontolni mennyi oktávot célszerű használni. Túl sok oktávnál előléphet a korábban említett túlindexelés is, mivel mindegyik oktáv lépésszáma az előzőének a fele, így ki lehet futni a gradiens vektorokból.

A Perlin-zaj különböző paramétereinek kipróbálása közben azt vettem észre, hogy nem érdemes öt oktáv fölé menni. Öt oktáv felett marginális lesz a hozzáadott értéke tesztmanőver szempontjából és közben lassabb is lesz a futási idő. Még az öt oktáv is lehet sok nagyobb frekvenciájú zaj esetén. Az oktávot tehát érdemes használni, a zaj frekvenciáját figyelembe véve. Különböző oktávok használhatóak továbbá különböző tesztesetek megvalósításához, ezért ezt a tesztmérnöknek van lehetősége állítani.

### Futási idő csökkentése

A gradiens vektoroknál és az oktávoknál is említettem az algoritmus futási idejét. Egy oktáv esetén például egy 10 000 x 10 000 elemű zaj generálása körülbelül tíz percet vesz igénybe. Két oktávnál ugyanezt ennek a kétszerese legenerálni. Ugyan 10 000 elem valószínűleg nem fog kelleni, de már 5 000 is 2 percig tart, ami 4 oktávnál 8 perc lenne.

Az algoritmus kétdimenziós, viszont tesztmanővert elég egydimenziós zajból készíteni. Csak három jel kell, egy kormányszög és kettő járműsebesség. A két járműsebesség szükségét később a [eloszlás címsor] pontban fogom kifejteni. Ezért egy 5 000x5 000 zajból elég csak három 5 000 elemű szeletet kivennem. Kiegészítettem az algoritmust egy olyan paraméterrel, amivel tetszőleges számú elemet ki lehet hagyni a generálásból. Azért nem elég például csak az első három elemet legenerálni és aztán azokkal folytatni, mert a Perlin-zaj értékei nem csak vízszintesen, de függőlegesen is közel vannak egymáshoz. Az első három 5 000 érték között kevés a változás, ami később a két különböző járműsebesség tekintetében gondot okozna.

A robusztussági tesztek általában éjszaka kerülnek futtatásra. Több óra futtatás alatt különböző vezetési helyzetek tesztelésére is van idő. Ezt a funkcionalitást figyelembe véve, több generált elemből több különböző helyzetet is lehet készíteni. Tehát nem lesz elég csak egy tesztmanőverhez szükséges zajt generálni.

Ezzel a változtatással drasztikusan csökkent az algoritmus futási ideje. A korábbi egy oktávos 10 000x10 000-es példa így 10 perc helyett, csak minden 100. elemet generálva 5 másodperc alatt elkészül. A 100 darab 10 000 elemű zajból pedig akár 33 különböző tesztesetet is lehet készíteni.

## esetleg 1D vs 2D

# Tesztmanőver készítés

A tesztmanőver elkészítését a korábban ismertetett Perlin-zaj algoritmussal fogom megvalósítani. Ez az algoritmus fogja a kormányszög és járműsebesség jelek alapját adni, amiket aztán módosítani kell a végleges tesztmanőver elérése érdekében. Minden tesztesethez három szeletet kell kivenni majd feldolgozni a legenerált zajból. Az egyikből kormányszög a másik kettőből pedig járműsebesség lesz.

## Folyamatábra

\*aktivitás diagram az algoritmusról\*

## Zaj tartományba tolása

A Perlin-zaj által generált értékek értékkészlete, mint korábban már meghatároztam -1 és 1. Ahhoz, hogy ebből kormányszög vagy járműsebesség legyen előbb meg kell változtatni az értékkészletét. Megfelelő tartományba tolni a jeleket azért fontos, hogy ne fordulhasson elő olyan szituáció, ahol például több a kormányszög, mint amennyit a tesztelendő autóban fizikailag ki lehetne tekerni. Ezek a határok az autó és szervokormány mechanikai felépítéséből adódnak, így gépjárműveként változnak. Fontos tehát ezeknek a határoknak a betartása a végpozícióra felfeszülés és a valóságtól elrugaszkodó tekerések elkerülése érdekében. A járműsebességet szimulációs modell álltja be, de így is van fizikai hatása a rendszerre. Ugyanis a később bemutatásra kerülő rack terhelés járműsebesség alapján működik. Ezért mindkét jelnél fontos lesz a megfelelő tartományba tolás. Ezt erre a két jelre különböző módon valósítottam meg.

A kormányszög egyszerűbb volt, mert negatív és pozitív értékek is előfordulnak vezetés során. Ezt a tesztmérnöknek kell beállítania egy szám megadásával. A kódban pedig elegendő felszorozni a meglévő zaj értékkészletét, hogy megkapjuk a kívánt értékeket. A zaj értékei a Perlin algoritmus miatt csak ritkán érik el a határértékeiket. Ez azonban a kormányszögnél nem probléma, mert a maximum értékét szeretnénk beállítani. Vezetési módtól, tesztelendő munkaponttól függően nem célszerű mindig csak a maximális kormányszöget megadni. Autópályás vezetésnél nagy járműsebesség mellett például elég csak -30 és 30 fok vagy még kisebb tartományban tekerni. Bár a végső algoritmus nem engedné, hogy nagy járműsebesség mellet túl nagy legyen a kormányszög, a jel viszont teljesen elveszítené az eredeti formáját. Ezáltal pedig egy valóságtól elrugaszkodó tesztmanővert kapnánk.

A járműsebességet a meghatározott munkaponttól függően állítottam be. Egy munkapontot pedig a járműsebesség tartománya határoz meg. Tehát a tesztmérnök állíthatja be milyen járműsebesség tartományban szeretne tesztmanővert készíteni. Ennek a megvalósítása nehezebb volt, mert itt a konkrét értékek közé kell a zajt eltolni. Továbbá negatív értékek értelmezhetetlenek a járműsebesség szempontjából.

Először a bekért értékek közötti különbséget számoltam ki, hogy aztán majd azzal felszorozva és a kisebbik számot hozzáadva tudjam a kért tartományba tolni. De előtte egy határétékeket nem érintő zaj esetét kellet kompenzálnom. Megnéztem a zaj abszolút minimumát és maximumát, aztán elosztottam egyet azzal amelyik közelebb volt az egyhez, majd az eredménnyel megszoroztam a zaj tömböt. Ezzel azt értem el, hogy a minimum vagy maximum értéke a megadott tartománynak tökéletesen legyen beállítva, míg a másik is közelebb legyen a kívánt értékhez. Ezután 0 és 1 közé toltam a zaj értékkészletét és végrehajtottam a korábban ismertetett műveleteket. Végül pedig megnéztem, hogy a kívánt tartomány minimumához vagy maximumához vannak közelebb a zaj szélsőértékei és eltoltam, hogy egyenlő távolságra legyenek a szélsőértékek az elemektől a tartományban. Ennek a procedúrának a segítségével a járműsebesség a megadott értékek és a zaj értékei között csak minimális eltérés lesz.

## Eloszlás változtatása

A Perlin-zaj eloszlása egy normális eloszláshoz hasonlít, ahol az értékek középen csúcsosodnak. Az értékkészletéből és a normális eloszlásra hasonlításából következik, hogy az átlag, vagyis az eloszlás központi része mindig 0-hoz közelít.

A képen diagram, szöveg, képernyőkép, Diagram látható

Automatikusan generált leírás

Perlin-zaj eloszlása

Ez ugyan a kormányszögnél egy jó tulajdonságnak mondható, mert a középpont ennél is 0 és minél messzebb van a középponttól annál valószínűtlenebb az érték megjelenése, de a járműsebességhez már nem ideális. A különböző vezetési módoknál normális eloszlással nem lehet elérni a kívánt működést. Autópályás vezetésnél nem valósághű, hogy egy nagyobb és kisebb, a középpontól ugyanannyira lévő sebesség ugyanazt a valószínűséget kapja. [\*Ha van, statisztika link ezzel kapcsolatban\*] Például autópályán nem feltétlenül egyenlő a 120 km/h és 80 km/h járműsebesség gyakoriságának a valószínűsége. A valósághoz közelebbi működés érdekében ezért, a Perlin-zajhoz tetszőleges eloszlást kell rendelni. A tesztmérnök így meghatározhatja, hogy az adott teszt milyen vezetési móddal fusson, milyen gyakran forduljanak elő adott értékek.

## Eloszlás változtatása 2

Az eloszlás tetszőleges beállításához két olyan jelre van szükség, amelyek más tartományban vannak. Ha az egyik jelből időközben átváltunk a másikba akkor elérhetünk egy normális eloszlástól eltérő jelet. Például, ha van egy jelünk 40 és 60 között és egy másik 90 és 130 között, akkor az ezekből készült jel eloszlását úgy szabhatjuk tetszésünkre, hogy az elsőnek és a másodiknak is csak egy részét használjuk fel. És az, hogy mennyi részét használjuk fel a jeleknek meghatározza, hogy milyen eloszlású lesz az elkészült jelünk.

A tetszőleges eloszlás megvalósításához a jelenlegi járműsebességtől eltérő Perlin-zajt kell generálnunk. Ezt a két jelet illesztem majd össze az eloszlás megváltoztatásához. A tesztmérnöknek egy olyan „dictionary-t” kell megadnia, amiben benne vannak a két járműsebesség határértékei és az, hogy milyen arányban szerepeljenek az elkészült jelben. A két jelet először a megadott határértékhez tolom, illetve aszerint skálázom. Aztán az egyikből másikba való átmenetet egy 0-tól 1-ig feltöltött tömb segítségével valósítom meg. A bemenetként megkapott arány szerint végig megyek a tömbön és az elsőt megszorzom az egyből kivont éppen aktuális elemmel, a másodikat pedig az aktuális elemmel szorzom, aztán összeadom az értéküket. Ezáltal a két jelet összetudom úgy kötni, hogy ne legyen hirtelen értékváltozás az összecsatolás helyén. A végeredmény egy olyan járműsebesség, amit a tesztmérnök bemenetei alapján tetszőleges eloszlással és határértékekkel hoztam létre.

A képen szöveg, képernyőkép, diagram, Diagram látható

Automatikusan generált leírás

Módosított Perlin-zaj eloszlása

## Kormányszög és járműsebesség kapcsolata

Ha a kormányszöget és járműsebességet végig külön kezelnénk akkor a végén olyan eredményeket kapnánk, amelyek bár külön-külön helyesek lennének, együtt nem fednének le egy élethű autóvezetést. Ezért össze kell kapcsolni a kettőt, hogy legyen hatásuk egymásra és a tesztmanőver összeségében realisztikus legyen.

Az összekapcsoláshoz egy tanulmányt [link] vettem alapul, ahol a kormányszöget mérték, adott járműsebességeknél városi közlekedés során. A mérések alapján egy limitet állítottak be a kormányszögre, ami alatt az autóvezető még kontrollálni tudja az autót. Ezt a limitet aztán egy tesztpályán verifikálták.

A járműsebességet vettem alapul és ahhoz igazítottam a kormányszöget. A kormányszög limitet tehát a tanulmány mérései alapján állítottam be, ezzel garantálva az algoritmus, a valós vezetéshez minél közelebbi működést.

A képen szöveg, képernyőkép, sor, Diagram látható

Automatikusan generált leírás

kormányszög limitálás járműsebesség alapján

## Kormányszög és járműsebesség szabályozása

Fontos, a gyorsulás és a kormány eltekeredésének sebessége is szabályozva legyen. A kormányszög és járműsebesség felskálázása után előfordulhat olyan, hogy túl nagy a két egymást követő érték közötti különbség. Túl nagy különbség esetén pedig eltérne egy valósághű autóvezetéstől és legrosszabb esetben kárt is tehetne a tesztpadokban. Ezért mindenképpen szükség van egy olyan függvényre is, ami tetszőleges szögsebességre és gyorsulásra tud szabályozni.

\*stwa kormányszög rate limit kép\*

## Kormányszög és járműsebesség rampelés

A Perlin-zaj algoritmus által készült zajok eleje és vége nem feltétlenül nulla, hanem egy ahhoz közeli szám. Amikor a zajt tartományba toljuk, ezek a kis különbségek kormányszögnél nagyobbak lesznek. A járműsebesség esetében pedig tartománytól függően általában nem is lesz közel a nullához. Ez olyan problémát vet fel, hogy tesztmanőver esetén hirtelen egy adott értékre állítódik a járműsebesség, ami teljesen eltér bármilyen igazi autóvezetéstől. Állásból nem indulhat el egy autó 60 km/h-val, fokozatosan gyorsít, hogy elérje a kívánt sebességet. Kormányszögnél hasonlóan egy +100 fokban induló kormánytekerés nem valósághű és a tesztpad esetében nagyon gyors kormányszög elfordulási sebességet idézne elő.

Ennek javítására

\*járműsebesség ramp kép\*

# Kiértékelés

## Tesztmanőver futtatása

## Futtatott tesztesetek

## Összehasonlítás jelenlegi robusztussági tesztekkel

## Következtetések levonása