Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Bosnyák Bence

MOZGÓ HANGFORRÁSOK SUGÁRZÁSÁNAK MODELLEZÉSE PEREMELEM-MÓDSZERREL

KONZULENS

Dr. Rucz Péter

BUDAPEST, 2023

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Bosnyák Bence**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2023. 06. 11.

...…….

Bosnyák Bence

Összefoglaló

Mozgó hangforrások sugárzásának modellezése számos probléma megoldásában segíthet. Használható többek között közlekedési zajok szimulálásához, terek hangterjedésének optimalizálásához vagy akár játékfejlesztésben, hangforrások modellezésére.

A dolgozat az erre használható akusztikai peremelem-módszerrel foglalkozik. Ez egy numerikus módszer, melynek előnye, hogy egyenes vonalú egyenletes mozgást végző kiterjedt testek sugárzásának modellezését teszi lehetővé.

Mozgó hangforrás esetén problémát okoz a Doppler-jelenség megvalósítása, és a hangforrás iránykarakterisztikájának figyelembevétele a térszámításban.

A dolgozat ismerteti a peremelem-módszert a Helmholtz-egyenletből és a Kirchhoff-Helmholtz integrálegyenletből levezetve, és megmutatja, hogy egy kiterjedt test felszínét, hogyan lehet elemekre bontani, majd ezekre az elemekre illeszteni egy Gauss-kvadratúrát. Végül a szakdolgozat bemutat egy elkészült programot, ami megvalósítja egy hangsebességnél nem gyorsabb hangforrás sugárzásának modellezését.

Abstract

Modeling the radiation of moving sound sources can help solve various problems. It can be used, among other things, for simulating traffic noise, optimizing the propagation of sound in spaces, or even for game development, modeling sound sources.

This thesis deals with the acoustic boundary element method that can be used for this purpose. It is a numerical method that allows for modeling the radiation of extended bodies performing straight-line uniform motion.

In the case of a moving sound source, implementing the Doppler effect and calculating the directional characteristics of the sound source pose a problem.

This thesis presents the boundary element method derived from the Helmholtz equation and the Kirchhoff-Helmholtz integral equation and shows how the surface of an extended body can be divided into elements and fitted with a Gaussian quadrature. Finally, the thesis presents a developed program that realizes the modeling of the sound radiation of a moving source, which is not faster than the speed of sound.

1 Bevezetés

Mozgó hangforrások sugárzásának vizsgálatával lehetőség van különböző források azonosítására, illetve számos felhasználási módja van, például közlekedési zajok modellezésében. A dolgozat a sugárzás peremelem-módszerrel történő számításával foglalkozik. Mozgó hangforrások hangterének számítása esetén figyelembe kell venni a Doppler-jelenséget és a hangforrás iránykarakterisztikáját is.

1.1 A szakdolgozat célja

A dolgozat bemutatja a peremelem-módszer használatát egy mozgó hangforrásra, az ezt implementáló Python programon keresztül. A szakdolgozat ismerteti az akusztikai peremelem-módszert a Helmholtz-egyenletből levezetve, kitérve a különbségekre álló és mozgó források között. Az elkészült implementációt részletesen bemutatom, értékelem a kapott eredményeket, és transzparens teszttel igazolom a program számításainak helyességét.

1.2 A szakdolgozat felépítése

A dolgozat elején ismertetem a peremelem-módszert. Bemutatom, hogy hogyan lehet egy parciális differenciálegyenletet megoldani, ha ismert a hozzá tartozó szabadtéri Green-függvény, majd ismertetem a Helmholtz-egyenletet és a Green-függvényét. Ebből az egyenletből levezetem a Kirchhoff-Helmholtz integrálegyenletet, amiből aztán a peremelem-módszer mátrixegyenletét. Ezután megmutatom, hogy ezeket az egyenleteket hogyan kell megváltoztatni egy mozgó hangforrás sugárzásának modellezéséhez.

Ezt követően megmutatom, hogy a geometria elemei fölött elvégzendő integrálások, hogyan számolhatók ki Gauss-kvadratúra segítségével, és kitérek arra is, hogy a peremelem-módszer mátrixainak főátlói hogyan számíthatók ki, és hogy miért van szükség arra, hogy külön kezeljük ezt az esetet.

Mivel az elkészült program Python nyelven íródott, ezután röviden bemutatom ezt a programozási nyelvet, és a program elkészítése során használt csomagokat. A fontosabb csomagok használatára példákat is mutatok, és ismertetem, hogy milyen szerepük van az elkészült programban.

Majd részletesen ismertetem az elkészült programot. A program bemenete egy geometriát leíró fájl. Mutatok példát egy ilyen fájl készítésére GMSH programmal, és arra, hogy a Python program hogyan értelmezi azt. Bemutatom, hogy a geometria fájlból kapott háromszögek adatait hogyan számolja ki a program, illetve azt, hogy ezeken a háromszögeken hogyan helyezi el a kiszámított kvadratúrapontokat. Majd ismertetem, hogy a peremelem-módszer mátrixait hogyan lehet feltölteni álló és mozgó hangforrás esetén.

Az ezt követő fejezetben ismertetem a transzparens tesztet, és hogy hogyan lehet a peremelem-módszer mátrixait ezzel a teszttel ellenőrizni a geometriától függetlenül.

Ezután a program futtatásával kapott eredmények elemzése következik. Értelmezem a kapott ábrákat, és magyarázatot adok a látott grafikákra.

Befejezésképpen arról lesz szó, hogy az elkészült Python programot milyen módon lehet továbbfejleszteni, úgy hogy optimalizáljuk mind a futásidőt, mind a memóriafelhasználást.

2 A peremelem-módszer ismertetése

Ebben a részben a peremelem-módszer (Boundary Element Method, továbbiakban gyakran BEM) Helmholtz-egyenletből történő levezetéséről lesz szó.

2.1 Szabadtéri Green-függvény

Az akusztikai hullámegyenlet egy parciális differenciálegyenlet (továbbiakban gyakran PDE), így analitikus megoldása csak nagyon egyszerű, idealizált esetekben létezik. Viszont ha ismert a PDE szabadtéri Green-függvénye, akkor a PDE megoldása helyett elég két függvény konvolúcióját kiszámolni [1]. Fontos, hogy a szabadtéri Green-függvénynél nem feltételezünk peremeket, hanem végtelen téren értelmezzük. Általánosan:

$$\mathcal{L}\{F(x, x_0)\} = -\delta(x - x_0).$$

Itt $\mathcal{L}$ a lineáris differenciáloperátor, $F(x, x_0)$ a lineáris PDE szabadtéri Green-függvénye (Fundamental solution), δ pedig a Dirac-delta. Tegyük fel, hogy meg akarjuk oldani a következő PDE-t:

$$\mathcal{L}\{u(x)\} = -g(x).$$

Ha ismert F szabadtéri Green-függvény, akkor:

$$u = F * g,$$

$$u(x) = \int F(x, x_0) g(x_0)\, \mathrm{d}x_0.$$

A szabadtéri Green-függvény $F(x, x_0)$ egy x_0-ban elhelyezett, egységnyi erősségű pontforrás válaszát jelenti x pontban [2].

2.2 Helmholtz-egyenlet

A következő PDE-t nevezzük Helmholtz-egyenletnek:

$$\nabla^2 p + k^2 p = -Q.$$

Az egyenletben k a hullámszám, tehát $k = \frac{2\pi}{\lambda}$, ahol λ a hullámhossz, Q pedig a források terének térbeli eloszlását leíró függvény. A nabla operátor (∇) jelentése $p(x, y, z)$ függvényre:

$$\nabla p = \frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}, \frac{\partial p}{\partial z}.$$

$\nabla^2 p = \nabla \cdot \nabla p$, tehát:

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}.$$

Vegyük észre, hogy ∇^2 miatt p függvénynek a térben mindenhol kétszer differenciálhatónak kell lennie.

2.3 A Helmholtz-egyenlet Green-függvénye

A Helmholtz-egyenlethez tartozó szabadtéri Green-függvény meghatározható Fourier-transzformációval, és háromdimenzióban az alábbi alakot ölti:

$$G(x, x_0) = \frac{e^{-jkr}}{4\pi r}.$$

A green-függvényben k a hullámszám és $r = |x - x_0|$.

A nyomás amplitúdójának $\frac{1}{r}$ csökkenése a teljesítmény $\frac{1}{r^2}$ csökkenését jelenti, vagyis r távolságra felvéve egy gömbfelületet a felületen átmenő lesugárzott teljesítmény r-től függetlenül konstans.

2.4 Kirchhoff-Helmholtz integrálegyenlet

Ebben a részben a Helmholtz-egyenletből levezetem peremelem-módszer alapjául szolgáló Kirchhoff-Helmholtz integrálegyenletet.

Előállítjuk a parciális differenciálegyenlet gyenge alakját, ami azt jelenti, hogy veszünk egy tetszőleges függvényt (ψ), úgynevezett tesztfüggvényt, és beszorozzuk vele az egyenletet, majd végig integráljuk az egyenletet a teljes megoldási tartományon.

$$\int_\Omega \psi(x)[\nabla^2 p(x) + k^2 p(x)]\, dx = \int_\Omega -\psi(x) Q(x)\, \mathrm{d}x$$

Fontos, hogy csak akkor lesz ekvivalens az eredeti PDE és a gyenge alak, ha ψ tetszőleges függvény lehet. Integráljuk parciálisan a gyenge alak $\psi\nabla^2 p$ tagját.

$$\psi\nabla^2 p = \nabla \cdot (\psi \nabla p) - \psi \nabla \cdot \nabla p = \nabla \cdot (\psi \nabla p) + \nabla \cdot (\nabla \psi p) + \nabla^2 \psi p$$

Írjuk vissza az integrálba.

$$\int_\Omega \nabla \cdot (\psi \nabla p)\,\mathrm{d}x - \int_\Omega \nabla \cdot (\nabla \psi p)\,\mathrm{d}x + \int_\Omega \nabla^2 \psi p\,\mathrm{d}x + \int_\Omega \psi k^2 p\,\mathrm{d}x = \int_\Omega -\psi Q\,\mathrm{d}x$$

A Gauss-tétel szerint térfogati integrálból felületi integrál készíthető, a következő módon: $\int_\Omega \nabla \cdot (f\bar{g})\,\mathrm{d}x = \int_\Gamma f\bar{g} \cdot \bar{n}\,\mathrm{d}x$, ahol felülvonás jelöli a vektorokat, és $\bar{n}$ a normálvektor, ami a felületen kifelé mutat. Alkalmazzuk ezt az első két tagra, és vonjuk össze a harmadik és negyedik tagot.

$$\int_\Gamma \psi \frac{\partial p}{\partial n}\mathrm{d}x - \int_\Gamma \frac{\partial \psi}{\partial n} p\,\mathrm{d}x + \int_\Omega [\nabla^2 \psi + k^2 \psi]\,\mathrm{d}x = \int_\Omega -\psi Q\,\mathrm{d}x$$

A korábbiakban kikötöttük, hogy p függvénynek a térben mindenhol kétszer differenciálhatónak kell lennie, viszont ebben a formában a differenciálegyenlet nem tartalmaz p-re kikötést. Ezért nevezzük ezt a PDE gyenge alakjának. Vegyük észre viszont, hogy $H\{\psi(x)\} = \nabla^2 \psi + k^2 \psi$, tehát a Helmholtz-operátor most a ψ próbafüggvényre hat, ennélfogva az eredetileg tetszőlegesen megválasztott ψ függvénynek kell a térben mindenhol kétszer differenciálhatónak lennie.

A következő lépésben $\psi \triangleq G(x, x_0)$, azaz legyen ψ a szabadtéri Green-függvény.

$$\int_\Gamma G \frac{\partial p}{\partial n}\mathrm{d}x - \int_\Gamma \frac{\partial G}{\partial n}\mathrm{d}x - \alpha(x_0)p(x_0) = \int_\Omega -GQ\,\mathrm{d}x$$

Itt $\alpha(x_0) = 1, \frac{1}{2}, 0$, attól függően, hogy x_0 a tartományon belül (Ω), a peremen (Γ) vagy azon kívül helyezkedik el. Fontos, hogy $\alpha = \frac{1}{2} \quad x_0 \in \Gamma$ csak akkor igaz, ha a perem sima. Ha a peremen van csúcs, akkor $\alpha = \frac{\omega}{4\pi}$ lesz, ahol ω a csúcs térszöge.

Rendezzük át az egyenletet.

$$\alpha(x_0)p(x_0) = \int_\Gamma G(x, x_0) \frac{\partial p(x)}{\partial n}\mathrm{d}x - \int_\Gamma \frac{\partial G(x, x_0)}{\partial n}\mathrm{d}x + \int_\Omega G(x, x_0)Q(x)\,\mathrm{d}x$$

Ezzel megkaptuk a Kirchhoff-Helmholtz integrálegyenletet.

Vizsgáljuk meg az integrálegyenlet tagjait.

$$\int_\Omega G(x, x_0)Q(x)\,\mathrm{d}x$$

Itt a Q forráseloszlás és a Green-függvény konvolúcióját láthatjuk, ami megadja a források terét azzal a feltételezéssel, hogy Ω tartomány végtelen. A másik két tag korrigálja ezt a véges tartomány figyelembevételével.

2.5 Peremelem-módszer

A célunk a Kirchhoff-Helmholtz integrálegyenletből egy algebrai egyenletrendszert létrehozni, mivel azt tudjuk számítógéppel megoldani. A továbbiakban így módosítjuk az integrálegyenletet:

$$\int_{\Omega} G(x, x_0) Q(x)\, \mathrm{d}x = 0\,.$$

Ezzel azt feltételezzük, hogy az Ω tartományon belül nem helyezkednek el térben elosztott források. A peremen elhelyezkedő forrásokat a másik két tag $\left(\int_{\Gamma} G(x, x_0) \frac{\partial p(x)}{\partial n} \mathrm{d}x - \int_{\Gamma} \frac{\partial G(x, x_0)}{\partial n} \mathrm{d}x\right)$ foglalja magába. A rezgésakusztikában a legtöbbször rezgő felületekről való lesugárzást, visszaverődést vizsgálunk, ezért gyakran élhetünk a $Q = 0$ feltétellel.

Következő lépés a geometriai diszkretizálás, amit a következőképpen írhatunk le:

$$\Omega = \bigcup_i \Omega_i \qquad \Omega_i \cap \Omega_j = \emptyset \qquad i \neq j\,,$$

tehát a tartományt felosztjuk E darab nem átlapoló elemre, és ezek fölött számoljuk ki az integrált. Legyen $\frac{\partial p}{\partial n} = q$. p és q diszkretizálását alakfüggvényekkel végezzük. Alakfüggvényként legtöbbször valamilyen alacsonyrendű polinomfüggvényt (pl. lineáris, kvadratikus) választunk az elemek fölött. Mivel p és q deriválhatóságára nincsen megkötésünk, ezért a legegyszerűbb választással az alakfüggvényt itt olyannak választjuk, ami egy elem fölött konstans.

$$\int_{\Omega} G(x, x_0) \frac{\partial p(x)}{\partial n} \mathrm{d}x = \sum_{i=1}^{E} \int_{\Omega_i} G(x, x_0)\, \mathrm{d}x\; q_i$$

Tipikusan a peremen vagy p, vagy q ismert. Az ismeretlen változó meghatározásához egyenletrendszert alkotunk kollokációs módszerrel. Ehhez x_0 legyen az elemek középpontjában. Ennek hatására $\alpha(x_0)$ mindig $\frac{1}{2}$ lesz és $p(x_0) = p_i$.

$$\frac{1}{2}p_i = \sum_{j=1}^{E} \int_{\Omega_j} G(x, x_i)\,\mathrm{d}x\; q_j - \sum_{j=1}^{E} \int_{\Omega_j} \frac{\partial G(x, x_i)}{\partial n}\,\mathrm{d}x\; p_j$$

A továbbiakban nevezzük el $\int_{\Omega_j} G(x, x_i)\,\mathrm{d}x$-t G_{ij}-nek, $\int_{\Omega_j} \frac{\partial G(x,x_i)}{\partial n}\,\mathrm{d}x$-t pedig H_{ij}-nek. Mivel $i = 1, 2, 3, \ldots, E$, ez E darab egyenletet jelent. Mátrix alakban:

$$\frac{1}{2}\bar{p} = \bar{\bar{G}}\bar{q} - \bar{\bar{H}}\bar{p}$$

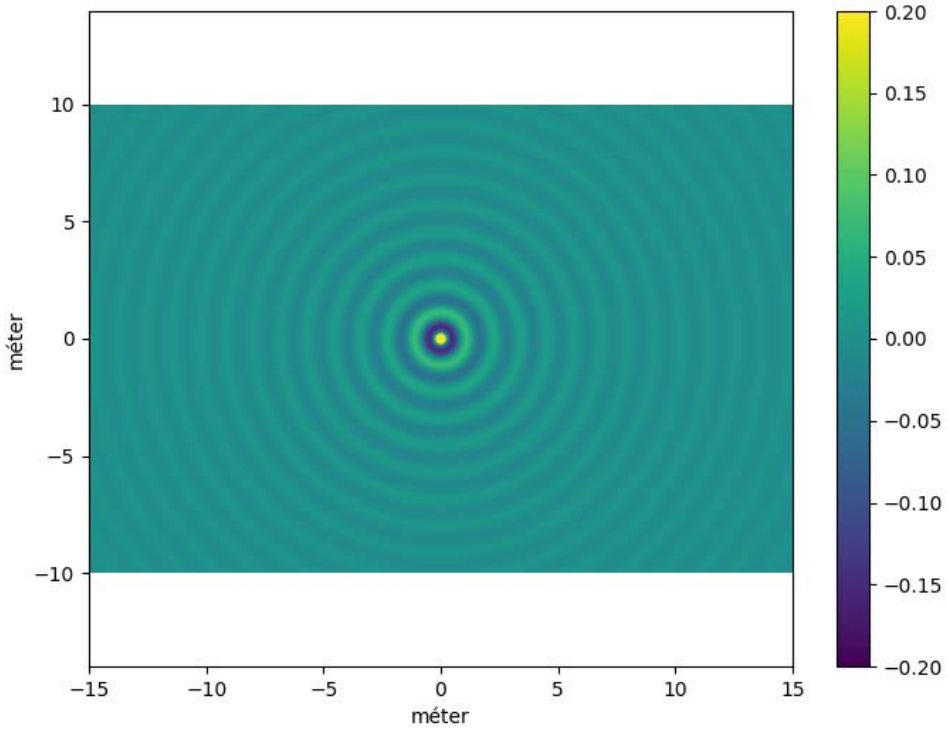
Ezzel megkaptuk a peremelem-módszer mátrixegyenletét, ahol felülvonás jelöli a vektorokat és kettős felülvonás a mátrixokat. $\bar{\bar{G}}$ és $\bar{\bar{H}}$ mátrixok frekvenciafüggőek, teliek és komplex értékűek lesznek.

Ha a felületen már ismert $\bar{p}$ és $\bar{q}$ vektor is, akkor az Ω tartomány tetszőleges pontjában ki tudjuk számítani p-t, újra alkalmazva az integrálegyenletet. $\alpha = 1$ lesz mindenhol, és az eredmény pedig mátrix-vektor szorzásként adódik.

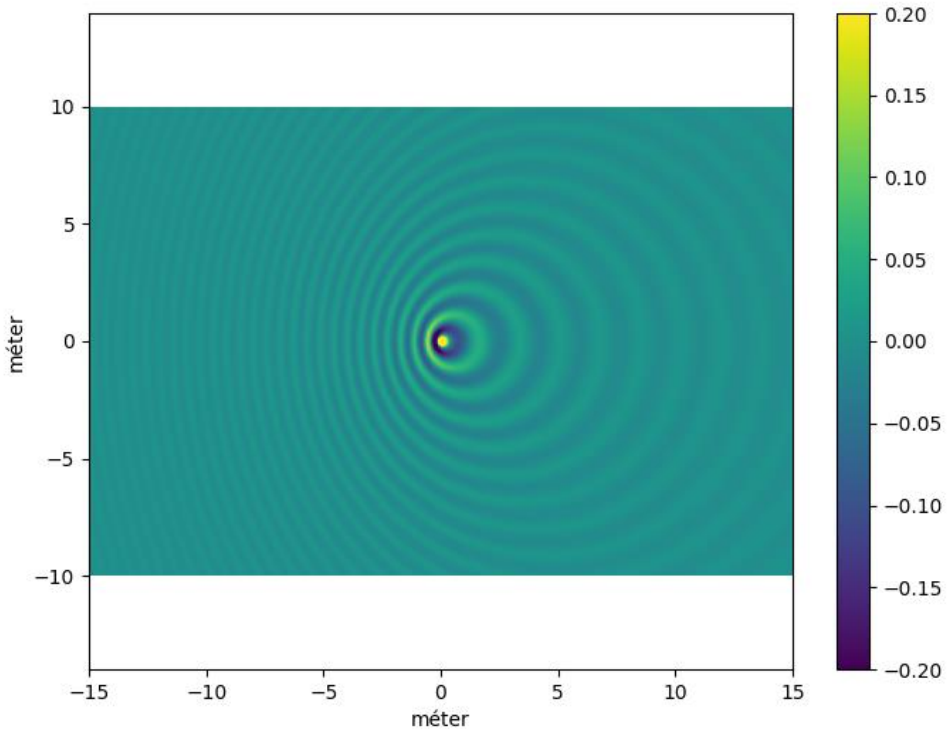
2.6 Mozgó hangforrás

A következőkben arról lesz szó, hogy hogyan kell változtatnunk az integrálegyenleteken a mozgó hangforrás terének számításához. Vegyük észre, hogy az a probléma, ahol a hangforrás egyenes vonalú egyenletes mozgást végez, és az a probléma ahol a közeg áramlik ekvivalens. A mozgó forrás álló vevőhöz lesugárzott terét úgy kapjuk, hogy a mozgó forrással együtt mozgó koordinátarendszerben egy, a mozgás irányába eső egyenes mentén számoljuk ki a lesugárzott teret.

Hasonlítsuk össze a hullámfrontokat álló (ld.: 2.1. ábra), illetve mozgó (ld.: 2.2. ábra) hangforrás esetén. Megfigyelhető, hogy a hullámfrontok mozgó hangforrásnál sűrűsödnek.

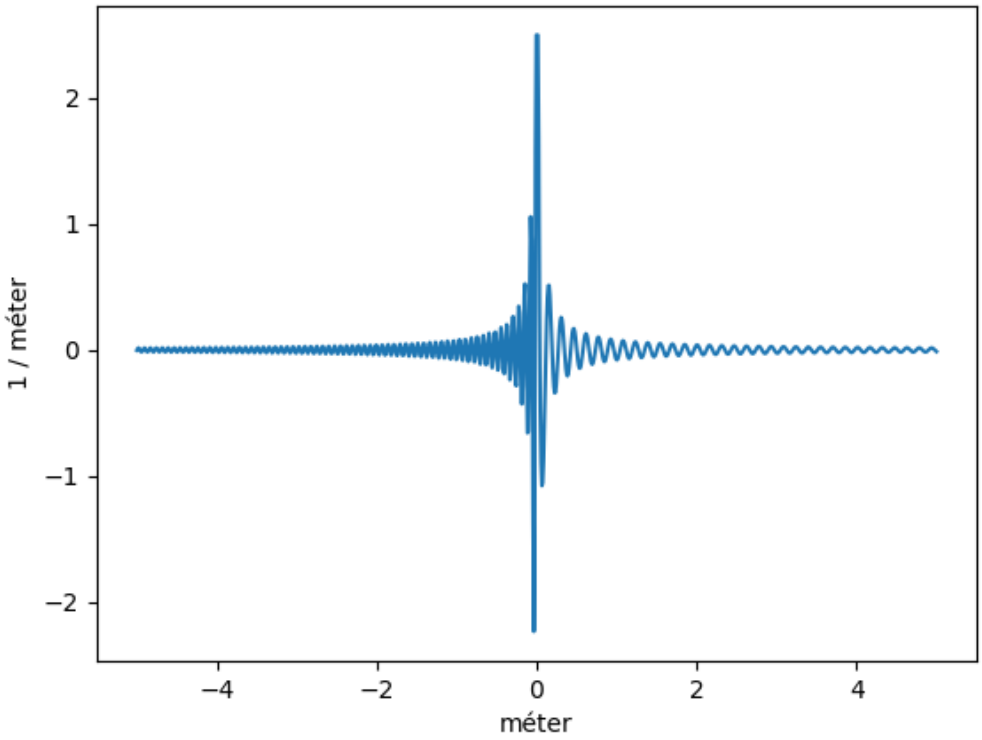


2.1. ábra: Hullámfrontok álló hangforrás esetén



2.2. ábra: Hullámfrontok mozgó hangforrás esetén

Ha ezeket az értékeket egy egyenes mentén kiolvassuk, akkor jobban szemléltethető a Doppler-jelenség. Ezt mutatja a 2.3. ábra.



2.3. ábra: Doppler-jelenség

Megfigyelhető, hogy amíg a hangforrás a vevő felé mozog a frekvencia magasabb lesz, mint amikor már elhagyta a vevőt.

2.6.1 Módosított Helmholtz-egyenlet

A hangforrás egyenes vonalú egyenletes mozgása esetén a Helmholtz-egyenletet a következő módon kell módosítani:

$$\nabla^2 p + k^2 p - 2jkM\frac{\partial p}{\partial x} + M^2\frac{\partial^2 p}{\partial x^2} = 0.$$

Az egyenlet feltételezi, hogy a mozgás x irányba történik és, hogy $M < 1$. M a Mach-szám, azaz $M = \frac{|\bar{v}|}{c}$, ahol c a hangsebesség. Figyeljük meg, hogy ha $M = 0$, tehát álló hangforrásról van szó, akkor a harmadik és negyedik tag kiesésével visszakapjuk az eredeti Helmholtz-egyenletet.

2.6.2 Módosított Green-függvény

A Helmholtz-egyenlet módosításával természetesen a hozzá tartozó Green-függvény is változik:

$$G = \frac{e^{-jK(R+B)}}{4\pi R}.$$

A Green-függvényben K, R és B a következőket jelölik:

$$K = \frac{k}{1-M^2},$$

$$R = \sqrt{(x-x_0)^2 + (1-M^2)((y-y_0)^2 + (z-z_0)^2)},$$

$$B = M(x-x_0).$$

Itt is megfigyelhetjük, hogy $M = 0$ esetén visszakapjuk az álló hangforrás Helmholtz-egyenletéhez tartozó Green-függvényt.

2.6.3 Módosított Kirchhoff-Helmholtz integrálegyenlet

A mozgó hangforrás Helmholtz-egyenletéből a 2.4-es fejezethez hasonlóan levezethetjük a Kirchhoff-Helmholtz integrálegyenletet. Tehát vesszük az egyenlet gyenge alakját, parciálisan integráljuk, majd a tesztfüggvény helyére behelyettesítjük az adjungált PDE Green-függvényét. Ha ezt megtesszük a következőt kapjuk:

$$\alpha(x_0)p(x_0) = \int_S \left(G\frac{\partial p}{\partial n} - p\frac{\partial G}{\partial n}\right) - 2jkMGpn_1 - M^2\left(G\frac{\partial p}{\partial x} - p\frac{\partial G}{\partial x}\right)n_1 \,\mathrm{d}S.$$

Itt n_1 a normálvektor x komponensét jelöli.

Az $\bar{x}$ vektort bontsuk fel egy normális és egy tangenciális irányú komponensre.

$$\bar{x} = (\bar{x}\cdot\bar{n})\cdot\bar{n} + (\bar{x}\cdot\bar{t})\cdot\bar{t}$$

$$\frac{\partial p}{\partial x} = (\bar{x}\cdot\bar{n})\frac{\partial p}{\partial n} + (\bar{x}\cdot\bar{t})\frac{\partial p}{\partial t}$$

Mivel $\bar{t}$ tangenciális irányú és a kiválasztott konstans alakfüggvények az elemen tangenciális irányban nem változnak $(\bar{x}\cdot\bar{t})\frac{\partial p}{\partial t} = 0$ lesz, tehát ezt a tagot elhagyhatjuk.

$$\frac{\partial p}{\partial x} = (\bar{x}\cdot\bar{n})\frac{\partial p}{\partial n} = n_1\frac{\partial p}{\partial n}$$

Ennek segítségével a Kirchhoff-Helmholtz integrálegyenlet így módosítható:

$$\alpha(x_0)p(x_0) = \int_S -\frac{\partial G}{\partial n}p + 2jkMGpn_1 + M^2\frac{\partial G}{\partial x}n_1 p \,\mathrm{d}S + \int_S G\frac{\partial p}{\partial n} - M^2{n_1}^2\frac{\partial p}{\partial n}\mathrm{d}S$$

A fentiekkel azonos $\frac{1}{2}\bar{p} = -\bar{\bar{H}}\bar{p} + \bar{\bar{G}}\bar{q}$ alakban felírható, ahol H_{ij} és G_{ij} a következőképpen számítható:

$$H_{ij} = \int_{\Omega j} \frac{\partial G(\bar{x_\iota}, \bar{x})}{\partial n} + 2jkMG(\bar{x_\iota}, \bar{x})n_1(\bar{x}) - M^2\frac{\partial G(\bar{x_\iota}, \bar{x})}{\partial x}n_1(\bar{x})\,\mathrm{d}\bar{x},$$

$$G_{ij} = \int_{\Omega j} G(\bar{x_\iota}, \bar{x})[1 - M^2{n_1}^2(\bar{x})]\,\mathrm{d}\bar{x}.$$

3 Integrálás az elemek fölött

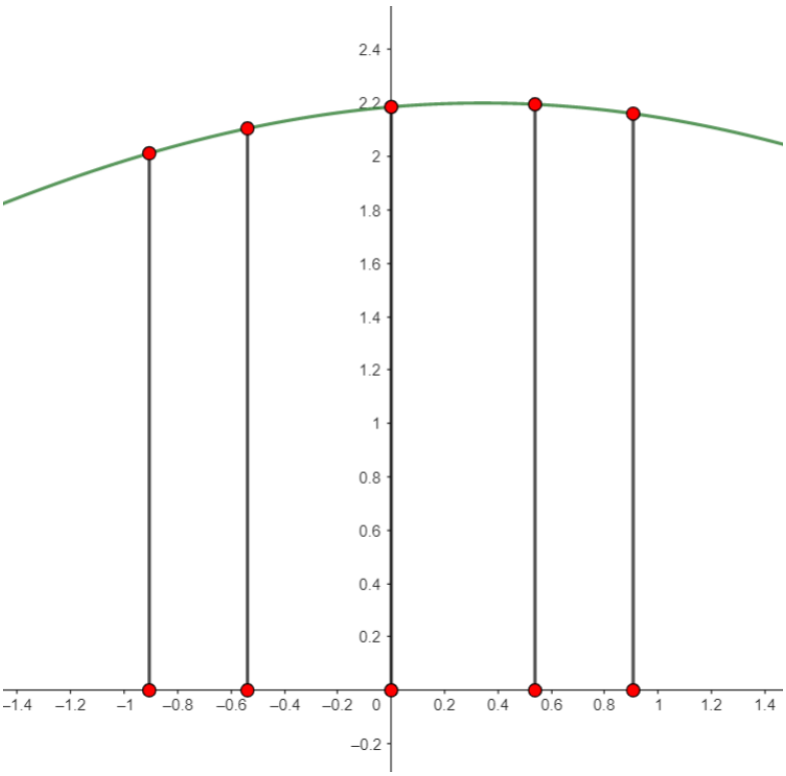
Az előzőekben nem kötöttünk ki semmit Ω_i tartományok alakjára, a gyakorlatban viszont a geometriánkat háromszögekre fogjuk osztani. A BEM mátrixok kiszámításához tehát a feladatunk Green-függvény és a Green-függvény normális irányú deriváltjának integrálása a háromszögek fölött.

3.1 Gauss-kvadratúra

A Gauss-kvadratúra lehetővé teszi egy integrál elvégzését, megfelelően megválasztott pontok és hozzájuk tartozó súlyok segítségével a következő módon:

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx \sum_{i=1}^{N} f(x_i)\,w_i,$$

ahol w_i az $f(x_i)$-hez tartozó súly és N a kvadratúra pontszáma, azaz azt jelenti, hogy hány ponttal osztjuk a tartományt. Egy N darab pontból álló kvadratúra pontos eredményt ad egy maximum $2N-1$-ed fokú polinomra. A pontok és súlyok számítására ez a dolgozat nem tér ki.



3.1. ábra: Gauss-kvadratúra pontok szemléltetése $N = 5$ esetén

Vegyünk egy egyszerű példát. Integráljuk az $f(x) = x^2$ függvényt a $-1, 1$ tartományon.

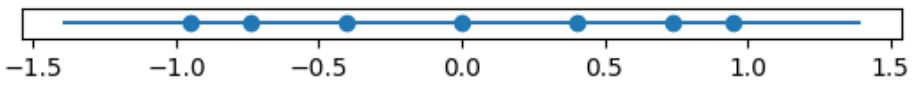
$$\int_{-1}^{1} x^2 \, \mathrm{d}x = \left[\frac{x^3}{3}\right]_{-1}^{1} = \frac{2}{3}$$

Ha Gauss-kvadratúrával szeretnénk elvégezni az integrálást, akkor elég a kvadratúra pontszámát 2-nek választani, mert az integrálandó polinom csak másodfokú. Ekkor a pontok $-\sqrt{\frac{1}{3}}, \sqrt{\frac{1}{3}}$ lesznek a súlyok pedig $1, 1$. Tehát azt mondhatjuk:

$$\int_{-1}^{1} x^2 \, \mathrm{d}x = \left(-\sqrt{\frac{1}{3}}\right)^2 \cdot 1 + \left(\sqrt{\frac{1}{3}}\right)^2 \cdot 1 = \frac{2}{3}.$$

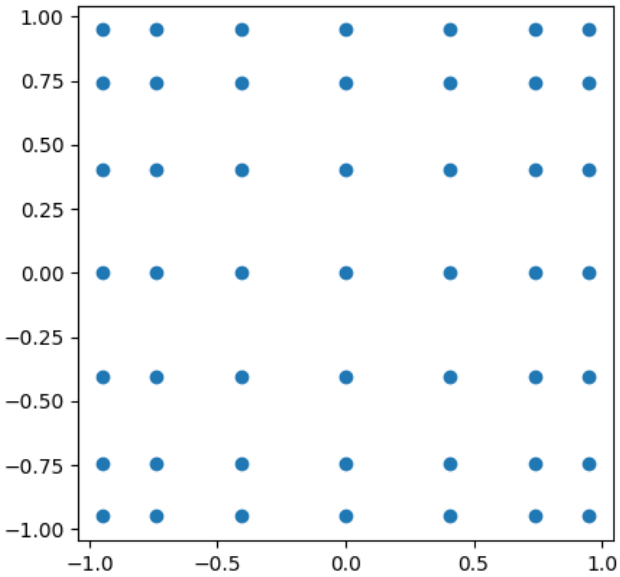
3.2 Kvadratúrapontok elhelyezése a háromszögeken

Jelenleg tetszőleges pontszámú kvadratúrát létre tudunk hozni a $-1, 1$ tartományon. A további ábrákon feltételezem, hogy $N = 7$, de természetesen a megoldás tetszőleges pontszámú kvadratúrára működik. A 3.2. ábra ezeket a kvadratúrapontokat szemlélteti a számegyenesen.



3.2. ábra: Kvadratúra pontok a számegyenesen

Ahhoz, hogy ezeket a pontokat rá tudjuk helyezni egy háromszögre, kétdimenziós kvadratúrapontokat hozunk létre úgy, hogy vesszük az eredeti kvadratúra pontok minden lehetséges párosítását, azaz a tenzorszorzatát. Ezzel alkotunk N^2 darab pontot. Ennek az eredményét mutatja a 3.3. ábra.



3.3. ábra: Kétdimenziós kvadratúrapontok

A pontok módosításával a súlyokon változtatnunk kell.

$$w_{i,j} = w_i \cdot w_j$$

Tehát egy kétdimenziós pont súlyát úgy kaphatjuk meg, ha összeszorozzuk az azt alkotó két eredeti kvadratúrapont súlyát. Így a súlyok összege egyenlő lesz néggyel, azaz a négyzet területével.

A következő lépésben rá fogjuk helyezni a kvadratúrapontokat egy háromszögre. Ehhez az eddigi négyzet $(-1,-1),(-1,1),(1,-1),(1,1)$ koordinátáit módosítjuk a következőképpen: $(0,0),(0,0),(1,0),(0,1)$, tehát úgy készítünk belőle háromszöget, hogy két koordinátáját megegyezővé tesszük.

Vegyük $P_0(\xi_{0,}\,\eta_0)$ kvadratúrapontot, és képezzük a következő mátrixot:

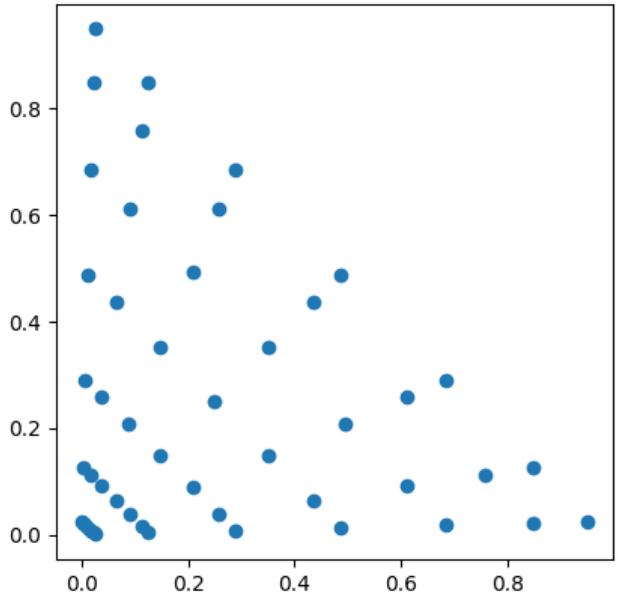
$$\left[\frac{(1-\xi_0)(1-\eta_0)}{4} \quad \frac{(1+\xi_0)(1-\eta_0)}{4} \quad \frac{(1+\xi_0)(1+\eta_0)}{4} \quad \frac{(1-\xi_0)(1+\eta_0)}{4}\right],$$

majd szorozzuk össze a háromszög koordinátáit tartalmazó mátrixszal és megkapjuk a pont koordinátáját a háromszögön.

$$\left[\frac{(1-\xi_0)(1-\eta_0)}{4} \quad \frac{(1+\xi_0)(1-\eta_0)}{4} \quad \frac{(1+\xi_0)(1+\eta_0)}{4} \quad \frac{(1-\xi_0)(1+\eta_0)}{4}\right]$$

$$\cdot \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \left[\frac{(1+\xi_0)(1+\eta_0)}{4} \quad \frac{(1-\xi_0)(1+\eta_0)}{4}\right]$$

Tehát azt mondhatjuk, hogy $P_0(\xi_0, \eta_0)$-ból képzett pont a háromszögön $P'_0\left(\frac{(1+\xi_0)(1+\eta_0)}{4}, \frac{(1-\xi_0)(1+\eta_0)}{4}\right)$ lesz. Ezeket a P' pontokat mutatja a 3.4. ábra.



3.4. ábra: Kvadratúrapontok háromszögön

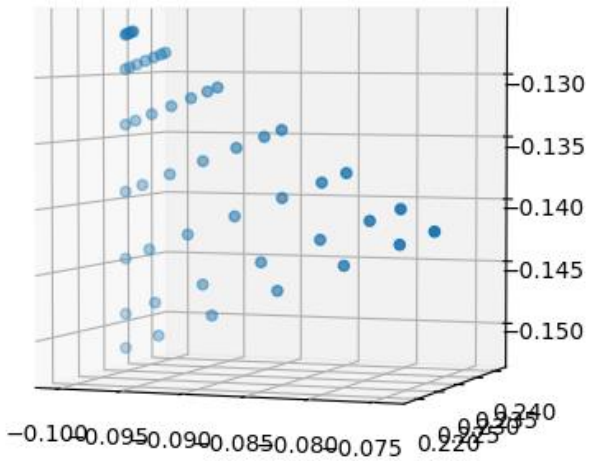
Természetesen a súlyokat ismét meg kell változtatni. $T_{négyzet} = 4$ volt, viszont $T_{háromszög} = \frac{1}{2}$, így a súlyok összege is $\frac{1}{2}$ kell, hogy legyen. A súlyok átskálázásához a fenti transzformáció Jacobi-determinánsát kell kiszámítani az egyes pontokban.

A következő lepés a kétdimenziós pontokat elhelyezni egy háromdimenziós háromszögön. Ehhez minden $P(\xi, \eta)$ pontokból képezzük $P(x, y, z) = (1 - \xi - \eta, \xi, \eta)$ pontokat. Így háromdimenziós pontjaink lesznek.

Ezeket a pontokat már csak rá kell helyezni a geometria háromszögeire. Ahhoz, hogy egy háromszögre rá tudjuk illeszteni a pontokat, vegyük a kvadratúrapontokat leíró N^2-szer hármas mátrixot és szorozzuk össze a háromszög pontjait leíró háromszor hármas mátrixszal. Így kapunk egy N^2-szer hármas mátrixot, ami tartalmazza a háromszögre

illesztett kvadratúrapontokat. Egy ilyen háromdimenziós háromszög kvadratúra pontjaira mutat egy példát a 3.5. ábra.

Ahhoz, hogy megkapjuk a helyes súlyokat minden súlyt meg kell szorozni a háromszög területének kétszeresével. Azért van szükség a kétszeres szorzóra, mert eredetileg a súlyok összege $\frac{1}{2}$, így ezután a művelet után lesz a súlyok összege egyenlő a háromszög területével.



3.5. ábra: Kvadratúrapontok a geometria egyik háromszögén

3.3 Az integrálás elvégzése

Integráljuk az adjungált PDE Green-függvényét (ld.: 2.6.2-es fejezet) a geometria egy tetszőleges háromszöge fölött. Vegyünk egy tetszőleges $P_0(x_0, y_0, z_0)$ pontot. A valós probléma megoldásakor P_0 a geometria háromszögeinek középpontjain fog végig iterálni, tehát minden háromszög középpontjából integrálunk minden háromszög fölött, és így töltjük fel a BEM mátrixokat. Az integrálás így végezhető el:

$$I = \sum_{i=1}^{N^2} \frac{e^{-jK(R_i+B)}}{4\pi R_i} \cdot w_i,$$

ahol N a kvadratúra pontszáma, w_i az i-edik kvadratúraponthoz tartozó súly és R_i a kvadratúrapontok és p_0 között így számítható:

$$R_i = \sqrt{(x_i - x_0)^2 + (1 - M^2)((y_i - y_0)^2 + (z_i - z_0)^2}.$$

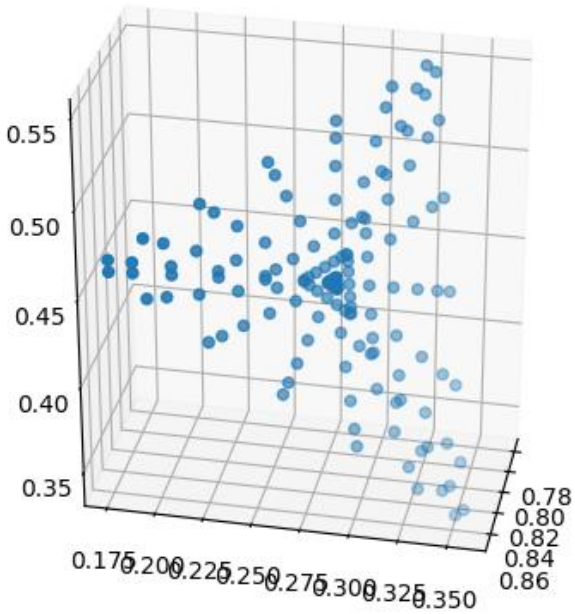
Tehát minden kvadratúrapontra ki kell értékelni a Green-függvényt, majd beszorozni a ponthoz tartozó súllyal, és ezeknek a szorzatoknak az összege fogja az integrálás eredményt megadni.

3.3.1 A BEM mátrixok főátlói

A BEM mátrixok feltöltésénél külön kezeljük azokat az eseteket, amikor a mátrixok átlóját számítjuk, azaz annak a háromszögnek a középpontjából integrálunk, ahol elhelyeztek a kvadratúrapontokat.

3.3.1.1 A G mátrix főátlója

Az adjungált PDE Green-függvénye csak akkor értelmezhető, ha $R \neq 0$, tehát ha egy kvadratúrapont sem esik a háromszög középpontjára. Ezt úgy tudjuk biztositani, hogy a háromszöget három részre osztjuk, majd ezekre a háromszögekre illesztjük a kvadratúrapontokat. Ezt mutatja be a 3.6. ábra.



3.6. ábra: Kvadratúrapontok a részekre osztott háromszögön

Így a kvadratúra pontok besűrűsödnek a középpont körül, viszont soha nem lesznek rajta.

Az integrálás eredménye, azaz a $\bar{\bar{G}}$ mátrix főátlójának eleme ilyenkor a három kisebb háromszög fölött kiszámított integrálok összege lesz.

3.3.1.2 A H mátrix főátlója

A $\bar{\bar{H}}$ mátrixba a Green-függvény normális irányú deriváltjának integráltjai kerülnek. Mivel a kvadratúrapontokból a háromszög középpontjába mutató vektorok és a háromszög normálvektora merőlegesek lesznek, kijelenthetjük, hogy egy álló hangforrás $\bar{\bar{H}}$ mátrixának főátlójának minden eleme nulla.

Mozgó hangforrás esetén a 3.3.1.1-es fejezetben ismertetett módszerrel kell kiszámítani az átló elemeit.

4 A Python programozási nyelv

A Python egy magas szintű, általános célú programozási nyelv, amelyet sok különböző probléma megoldására lehet használni [3]. Az elkészült program készítéséhez azért választottam, mert szintaxisa könnyen érthető és olvasható, és számos olyan könyvtár és modul íródott ehhez a nyelvhez, amely nagyban megegyszerűsítette a munkámat.

A Python programozási nyelvnek igen kiterjedt és széleskörű standard könyvtára van, amit még kiegészítenek az egyéb (mások által) megírt publikus modulok [4]. A továbbiakban az utóbbiakról lesz szó.

4.1 Pip

A pip a Python nyelv csomagtelepítője. A pip segítségével lehet csomagokat telepíteni a Python Package Index-ről (PyPI) és más indexekről [5].

A pip telepíthető a következő Windows Command Prompt-ban kiadott paranccsal:

```
python3 -m ensurepip --upgrade
```

Ha a pip telepítve van, akkor a különböző csomagok így installálhatók:

```
python3 –m pip install <csomagnév>
```

Ebben az esetben a megadott csomag legfrissebb verziója lesz telepítve. Lehetőség van a csomagok egy adott verziójának installálására is. Erre egy példa:

```
python3 -m pip install black==22.12.0
```

Ez a parancs a black csomag 22.12.0-ás verzióját telepíti.

4.2 NumPy

A NumPy az egyik alapvető csomag Pythonban a tudományos számításokhoz [6]. Az elkészült program futási ideje szempontjából kritikus ez a könyvtár, hiszen a műveletek a NumPy által biztosított mátrixokkal és vektorokkal, nagyságrendekkel gyorsabbak tudnak lenni, mint a ciklusokkal elvégzett számítások.

4.2.1 NumPy tömbök

A NumPy biztosít többdimenziós tömböket. Míg egy Python listában különböző adattípusok szerepelhetnek, egy NumPy tömbben az összes elemnek azonos típusúnak kell lennie, mivel így a tömbökön végezhető műveletek jóval hatékonyabbak tudnak lenni [7].

A Numpy könyvtár használatához a telepítés után importálni kell azt. Erre a konvenció az, hogy az importált nevet így rövidítjük:

```
import numpy as np
```

Importálás után NumPy tömböt a következő módokon hozhatunk létre:

```
array_1 = np.array([[1, 2, 3], [4, 5, 6]])
array_2 = np.empty((2, 3), dtype=int)
array_3 = np.zeros((2, 3), dtype=int)
```

Az első sor egy *(2, 3)* alakú tömböt hoz létre, amely így néz ki:

```
[[1 2 3]
 [4 5 6]]
```

A második sor szintén egy *(2, 3)* alakú tömböt hoz létre, amelyben talán azt gondolhatnánk, hogy nullák lesznek, de valójában inicializáltalan memóriaszemét van. Egy lehetséges kimenet:

```
[[ 96 116 112]
 [ 24 101  93]]
```

A harmadik sor hoz létre olyan *(2, 3)* alakú tömböt, amelynek minden eleme 0. Ezzel megegyező módon az *„np.ones()”* függvény olyan tömböt ad vissza, amelynek minden eleme 1.

Még egy függvény, amit használok a programban, és szintén NumPy tömböt ad vissza, így néz ki:

```
eye = np.eye(5, dtype=int)
```

Ez a függvény egy *(m, m)* alakú egységmátrixot ad vissza, ahol *„m”* a függvény első paramétere. A fenti kód eredménye:

```
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

4.3 Matplotlib

A Matplotlib könyvtár statikus, animált és interaktív vizualizációk létrehozásához használható Pythonban [8]. Matplotlib segítségével jelenítettem meg a programban használt geometriát és egyéb vizualizálható adatokat.

Egy egyszerű példa a könyvtár használatára:

```
import matplotlib.pyplot as plt

x, y = coords.T
plt.scatter(x, y, marker="D", color="red")
plt.show()
```

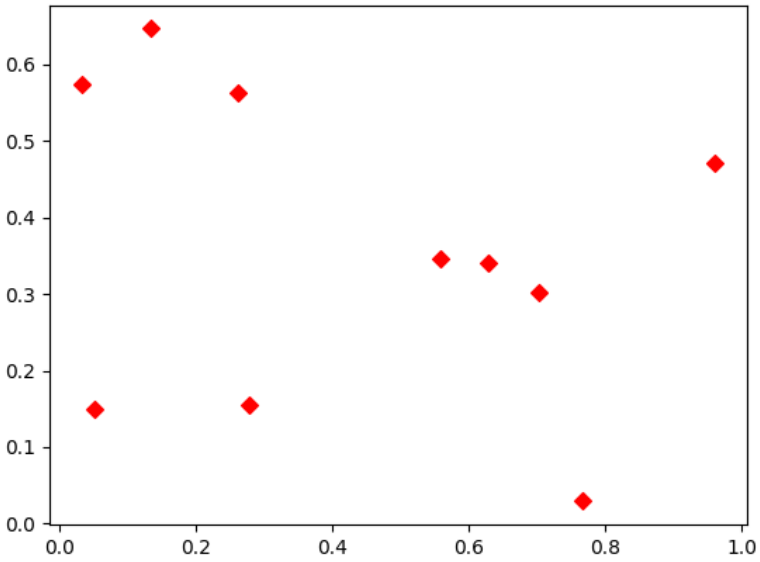
A fenti kódrészlet egy *(p, 2)* alakú mátrixban eltárolt, p darab kétdimenziós pont vizualizációját végzi el. Egy lehetséges példa a kódrészletben szereplő *„coords”* NumPy tömbre tehát:

```
[[0.76557345 0.02942652]
 [0.26166034 0.56212574]
 [0.13337506 0.64638068]
.
.
.
 [0.96084924 0.47003485]
 [0.62828057 0.33998496]
 [0.05096809 0.15029321]]
```

A fenti mátrix minden sora egy pont x és y koordinátáját tartalmazza. Természetesen a mátrix transzponáltja, így két sorból fog állni, és a kódrészlet második sora ezt a két sort tárolja el az *„x”* illetve az *„y”* nevű változókban. Ezzel minden pontnak az x, illetve y koordinátáját fogja tartalmazni a két változó és mindkettő tömb hossza meg fog egyezni a pontok számával.

A harmadik sorban a *„scatter()”* függvény kirajzolja az *„x”* és *„y”* változók által meghatározott pontokat. Az opcionális *„marker”* és *„color”* paraméterek miatt a pontokat piros színű gyémánt (Diamond) alak fogja jelölni.

Végül a *„show()”* függvény megjeleníti az elkészült ábrát. A 4.1. ábra egy ezzel a kóddal létrehozott ábrára mutat példát.



4.1. ábra: A pontok megjelenítve Matplotlib könyvtár segítségével

4.4 Egyéb használt csomagok

A továbbiakban a felhasznált, de kevésbé lényeges csomagokról lesz szó.

4.4.1 Black

A Black egy kódformázó a Python programozási nyelvhez. A program készítése során végig ezt használtam, mert sokkal egyszerűbb és gyorsabb, mint kézzel formázni, és a Black futtatása után nagyon letisztult és olvasható kódot kaptam. A Black a következő paranccsal futtatható.

```
python3 -m black thesis.py
```

Ezzel formázhatjuk a *„thesis.py”* modult.

4.4.2 Pylint

A Pylint futtatás nélkül elemzi a kódot. Hibákat keres és ellenőrzi a kódolási konvenciókat, majd javaslatokat tesz a problémák javítására [9]. Ez a csomag segített abban, hogy a kódomban ne legyenek nem használt változók, és egyéb kisebb hibákat is segített kijavítani.

5 Az elkészült program ismertetése

5.1 Geometria

Az elkészült Python program egyik bemenete egy fájl, ami leírja a geometriát. Ennek a fájlnak meg kell felelnie bizonyos formai előírásoknak, amiket ez az 5.1.2-es fejezet tárgyal.

5.1.1 Geometria létrehozása

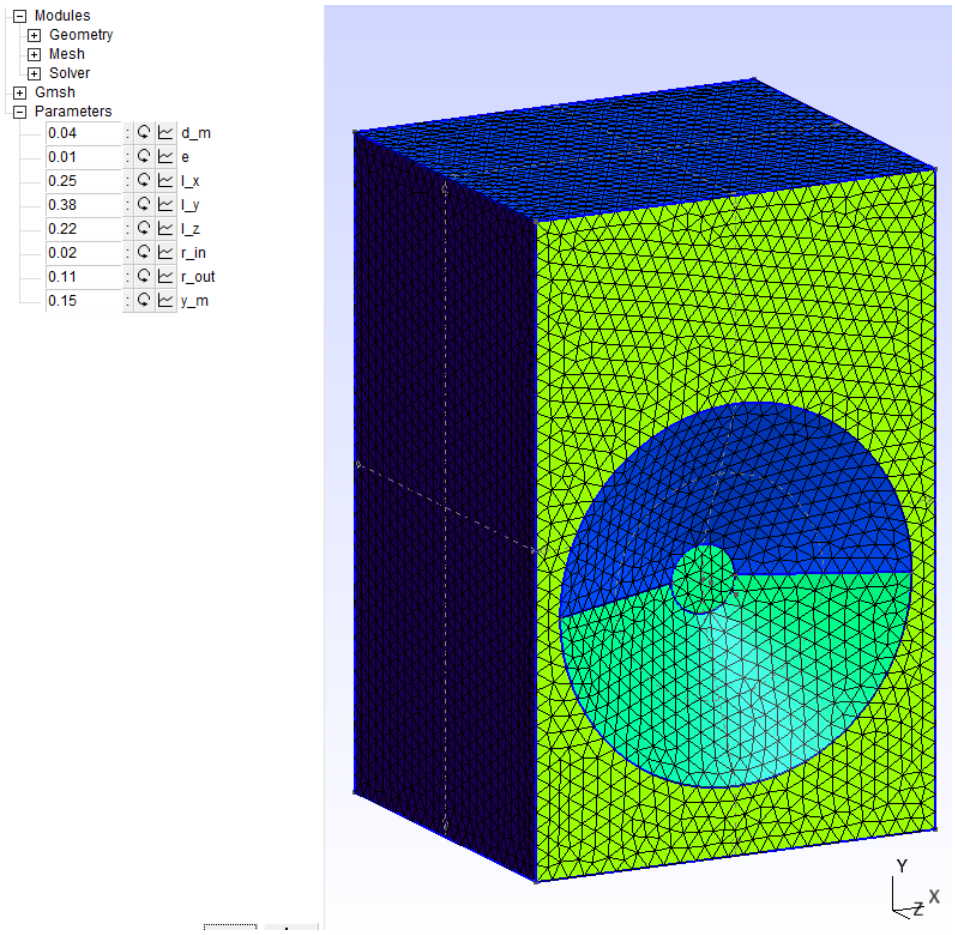
A használt geometriát a GMSH program használatával hoztam létre. A GMSH egy nyílt forráskódú szoftver, amely segítségével létrehozható egy olyan háromdimenziós geometria, amelyet az elkészült Python program bemenetként vár.

Egy ilyen geometria elkészítésének első lépése a testet meghatározó pontok, vonalak és felületek leírása, és paraméterek meghatározására a GMSH saját szkriptnyelvének használatával. Az így létrehozott *„.geo”* kiterjesztésű fájl felhasználásával dolgozhatunk.

A következő lépés a test felszínének feldarabolása tetszőleges méretű háromszögekre. Ezt a GMSH automatikusan elvégzi. Minél kisebb háromszögeket használunk, annál pontosabban közelítjük a *„.geo”* fájlban leírt alakzatot, viszont exponenciálisan növekszik a Python program memóriaigénye és futási ideje.

Az 5.1. ábra mutat egy példát a GMSH felhasználói felületére. Látható, hogy a bal oldali *„Parameters”* résznél lehetőség van a geometria paramétereinek módosítására. Leginkább ezek közül az ennél a geometriánál *„e”* névvel ellátott paraméter a legfontosabb, hiszen ez adja meg, az elemméretet, azaz hogy mekkora háromszögekre bontsa a GMSH a geometria felszínét.

Végül az így kapott háromszögekből álló testet egy *„.msh”* kiterjesztésű fájlba kell menteni. Ennél a lépésnél fontos figyelni, hogy a program csak akkor tudja megfelelően értelmezni a kapott geometriát, ha *„Version 2 ASCII”* formátumban van az elmentve.



5.1. ábra: Geometria készítése GMSH programmal

5.1.2 Geometria beolvasása

A megadott geometria fájl beolvasásáért a *„MeshReader”* osztály felelős. Az osztály konstruktora a geometriát meghatározó háromszögeket leíró *„.msh”* fájl elérési útvonalát várja paraméterként. A létrehozott objektum *„get_triangles()”* metódusa visszaadja a háromszögeket tartalmazó *(n, 3, 3)* alakú NumPy array-t, ahol *n* a háromszögek száma.

A geometriát tartalmazó fájlban vannak pontok, melyeket három-három valós koordináta határoz meg, illetve vannak a háromszögek, amelyek ezekre a pontokra tartalmaznak hivatkozásokat. Egy példa egy *„.msh”* fájl felépítésére:

```
$MeshFormat
2.2 0 8
$EndMeshFormat
```

A fájl elején található a formátum a *„$MeshFormat”* és *„$EndMeshFormat”* sorok között. A 2.2 a verziószám, a 0 azt jelzi, hogy ASCII karakterkódolás van

használva, míg a 8 azt jelenti, hogy a lebegőpontos számok 8 bájton vannak eltárolva. Ez a rész lényegtelen a Python programnak, így beolvasásnál ezt a részt átugorjuk.

```
$Nodes
63
1 0 0 0
2 -0.125 0.23 0
3 0.125 0.23 0
4 0.125 0.23 -0.22
5 -0.125 0.23 -0.22
.
.
.
59 0.03349707337812233 -0.04025750017071325 -0.02561289481288792
60 -2.995179881892796e-10 0.1521976308991798 0
61 -0.06803915583475734 0.1596500593249714 0
62 0.06803915526497087 0.1596500596409137 0
63 0 -5.782411586589357e-19 -0.04
$EndNodes
```

A *„$Nodes”* és az *„$EndNodes”* sorok között találhatóak a pontok. Az első sor jelzi, hogy 63 pontot tartalmaz a geometria, és a többi sorban vannak ezek pontok felsorolva. Minden ponthoz tartozik egy azonosító szám illetve három valós koordináta.

```
$Elements
167
1 15 2 0 0 1
2 15 2 0 1001 2
3 15 2 0 1002 3
4 15 2 0 1003 4
.
.
.
18 1 2 0 100 3 18
19 1 2 0 100 18 4
20 1 2 0 101 4 19
21 1 2 0 101 19 5
.
.
.
80 2 2 0 11 19 5 51
81 2 2 0 11 20 2 51
82 2 2 0 11 2 21 51
83 2 2 0 11 4 19 52
.
.
.
164 2 2 0 61 47 15 63
165 2 2 0 61 48 49 63
166 2 2 0 61 49 14 63
167 2 2 0 61 46 47 63
$EndElements
```

Az *„$Elements”* és az *„$EndElements”* sorok között találhatóak a geometria „elemei”. Ezek lehetnek vonalak, háromszögek vagy síkok, viszont a program csak a

háromszögeket olvassa be. A fenti kódrészletben azok a sorok tartalmaznak háromszögeket, ahol a sor második száma, az azonosító szám után kettes. Az ilyen sorok utolsó három eleme a háromszöget meghatározó három pont azonosítója. Ennek alapján például a *„81 2 2 0 11 20 2 51”* sor egy olyan háromszöget tartalmaz, amelynek azonosító száma *„81”* és a húszas, a kettes és az ötvenegyes azonosító számú pontok alkotják a csúcsait.

A MeshReader osztály konstruktora a paraméterül kapott elérési úton található fájlból beolvassa a pontokat egy *(k, 3)* alakú, valós számokat tartalmazó numpy tömbbe, ahol *k* a pontok számát jelöli és a háromszögeket egy *(n, 3)* alakú, egész számokat tartalmazó numpy tömbbe, ahol *n* a háromszögek számát jelöli. A háromszögeket tartalmazó tömb, tehát nem tartalmazza a csúcspontok koordinátait csak referenciákat a pontok tömb elemeire.

```
def __init__(self, mesh_file):
    self.nodes = np.empty((0, 3), float)
    self.triangles = np.empty((0, 3), int)

    with open(mesh_file, mode="r", encoding="utf-8") as f:
        line_type = None

        for line in f.readlines():
            if line.strip() == "$Nodes":
                line_type = "node"
                continue
            elif line.strip() == "$Elements":
                line_type = "element"
                continue
            elif line.strip() in ["$EndNodes", "$EndElements"]:
                line_type = None
                continue
            elif not line_type:
                    continue

            line = line.strip().split()

            if line_type == "node" and len(line) == 4:
                self.nodes = np.append(
                    self.nodes,
                    [[float(line[1]), float(line[2]), float(line[3])]],
                    axis=0,
                )
            elif line_type == "element" and len(line) == 8:
                self.triangles = np.append(
                    self.triangles,
                    [[int(line[5]), int(line[6]), int(line[7])]],
                    axis=0,
                )
```

Az osztálynak két attribútuma van. A *„nodes”* tárolja a három valós szám (float) által meghatározott pontokat, míg a *„triangles”* tömbben a három egész szám (int), azaz a *node*-ok sorszámai, által meghatározott háromszögek vannak.

Miután megnyitjuk a geometriát tartalmazó fájlt, végig iterálunk a sorokon. A *„line_type”* névű változóban eltároljuk azt, hogy a sor tartalma pont (*node*), elem (*element*) vagy egyik sem (*None*). A *„line_type”* kezdetben *„None”* és ha elérjük a pontok kezdetét jelző *„$Nodes”* vagy az elemek kezdetét jelző *„$Elements”* sort akkor megváltoztatjuk a változó értékét. Ha elérjük a pontok vagy elemek végét jelző *„$EndNodes”* vagy *„$EndElements”* sort akkor a változó értéke újra *„None”* lesz.

Ha a *„line_type”* értéke *„None”* akkor az adott sort nem olvassuk be. Ha *„node”* akkor a *„nodes”* listához hozzáadjuk a NumPy *„append()”* függvényével a sor azon részeit, amik a koordinátákat tartalmazzák. Vegyük észre, hogy a fenti logika szerint minden sor a *„$Nodes”* és *„$EndNodes”* sorok között pontot tartalmaz. Ez nem lesz igaz, hiszen az első sor a pontok számát tartalmazza, ezért szükséges lesz a sort alkotó, szóközzel elválasztott adatok számosságára feltételt kikötni.

Ha a *„line_type”* értéke *„element”*, és a sort nyolc adat alkotja, akkor a pontok sorszámait a *„triangles”* NumPy tömbhöz adjuk.

Mivel nehezebb lenne úgy használni az adatokat, hogy a háromszögek listája csak referenciákat tartalmaz pontokra, a létrehozott objektum *„get_triangles()”* metódusa egy olyan *(n, 3, 3)* alakú NumPy tömböt ad vissza, ahol *„n”* a háromszögek száma, és minden háromszöget három darab, valós számokat tartalmazó háromdimenziós koordináta határoz meg.

```
def get_triangles(self):
    triangles = []

    for triangle in self.triangles:
        triangles.append(
            [
                self.nodes[triangle[0] - 1],
                self.nodes[triangle[1] - 1],
                self.nodes[triangle[2] - 1],
            ]
        )

    return np.array(triangles)
```

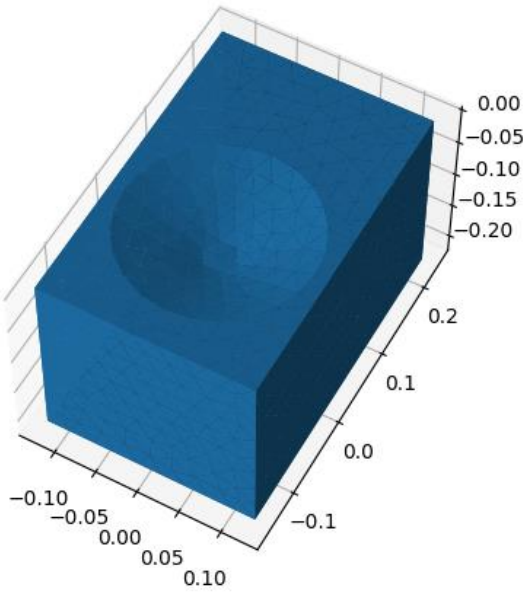
Ez a függvény egyszerűen végig iterál az osztály *„triangles”* attribútumán (nem összetévesztendő a függvény szintén *„triangles”* névű változójával), és a háromszögek

csúcspontjait meghatározó *„node”* sorszámokat, a megfelelő *„node”*-okban tárolt koordinátákra cseréli. A program a továbbiakban az így kapott *„triangles”* tömbbel dolgozik.

Egy példa a *„MeshReader”* osztály használatára:

```
from mesh_reader import MeshReader

mesh_file = r"C:\Users\User\Desktop\speaker.msh"
mesh_reader = MeshReader(mesh_file)
triangles = mesh_reader.get_triangles()
```

Lentebb az 5.2. ábra szemlélteti a fenti kódrész által létrehozott *„triangles”* NumPy tömböt. Láthatjuk a 2.5-ös fejezetben leírtakat, azaz, hogy ezeknek a nem átlapoló háromszögeknek az uniója valóban megadja a geometriát.



5.2. ábra: A triangles NumPy tömb megjelenítve

5.2 Háromszögek adatainak számítása

A BEM mátrixok számításához szükség van az összes háromszög területének, normálvektorának és középpontjának megtalálására. Ezt a célt szolgálja a *„get_triangle_data()”* függvény.

```
def get_triangle_data(triangles):
    normals = np.cross(
        triangles[:, 1, :] - triangles[:, 0, :],
        triangles[:, 2, :] - triangles[:, 0, :]
    )

    areas = np.linalg.norm(normals, axis=1) / 2
    normals = normals / np.linalg.norm(normals, axis=1, keepdims=True)
    centerpoints = np.mean(triangles, axis=1)

    return centerpoints, areas, normals
```

A függvény paraméterül kapja az *(n, 3, 3)* alakú triangles numpy tömböt, ahol *n* a háromszögek száma.

A normálvektorok meghatározásához minden háromszög két oldalának vektoriális szorzatát kell venni. Egy így kapott normálvektor merőleges a háromszög síkjára és hossza definíció szerint $|a| \cdot |b| \cdot \sin\gamma$, ahol ebben az esetben a és b a háromszög oldalai és γ a két vektor által bezárt szög.

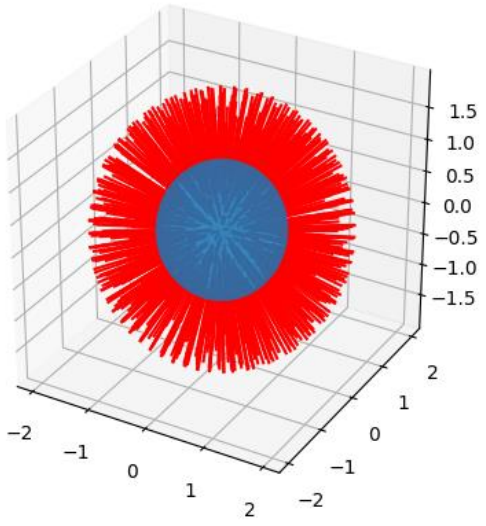
A trigonometrikus területképlet szerint egy tetszőleges háromszög területe kiszámolható a $T = \frac{a \cdot b \cdot \sin\gamma}{2}$ képlettel. Ezt a számítást láthatjuk az *„areas = np.linalg.norm(normals, axis=1) / 2”* sorban, ahol az NumPy *„linalg.norm()”* függvénye visszaadja a normálvektorok hosszát, majd mindegyik vektor hosszát osztjuk kettővel.

A kapott területek pontosságát könnyen ellenőrizhetjük, ha összehasonlítjuk a területek összegét a geometria felszínével. Tegyük fel, hogy a geometriánk egy egység sugarú gömb. Vizsgáljuk meg a következő két sort.

```
sum(areas)
4 * np.pi
```

Természetesen a két szám nem lesz egyenlő, hiszen a háromszögeknek csak a csúcspontjai lesznek a gömb felszínén, viszont kellően sok háromszög esetén olyan számot kell kiadnia a területek összegének, ami alig kisebb, mint a gömb valódi felszíne.

Mivel a későbbi számításokhoz egység hosszú normálvektorokra van szükség, ezért a *„normals = normals / np.linalg.norm(normals, axis=1, keepdims=True)”* sorban minden normálvektort elosztunk a saját hosszával.



5.3. ábra: Megjelenített normálvektorok egy egységgömb geometriához

A középpontok megtalálásához egyszerűen minden háromszög csúcspontjainak koordinátait átlagolni kell. Ezt csinálja a numpy *„mean()”* függvénye.

A függvény visszaadja az elkészült *(n, 3)* alakú *„centerpoints”*, az *(n, 1)* alakú *„areas”* és az *(n, 3)* alakú *„normals”* NumPy tömböket.

5.3 Kvadratúra pontok és súlyok számítása

Az elkészült program *„get_triangle_quadrature()”* függvénye a 3.2-es fejezetben leírtakat valósítja meg, azaz a függvény kívánt pontszámú Gauss-kvadratúra háromdimenziós pontjait és súlyait adja vissza, amelyeket megszorozva a geometria egy tetszőleges háromszögét leíró mátrixszal, megkaphatjuk a kvadratúrapontokat az adott háromszögön.

A következőkben ismertetem a függvényt, amit a hossza miatt részletenként fogok bemutatni. Pontok (…) jelzik a függvény részeinek határait, és a kódrészek alatt részletesebb magyarázatot adok a program működésére.

```
def get_triangle_quadrature(order):
    [points, weights] = np.polynomial.legendre.leggauss(order)
    …
```

A függvény paramétere a kvadratúra pontszáma. A NumPy *„polynomial.legendre.leggauss()”* függvénye visszaadja a Gauss-kvadratúra pontjait és súlyait a paramétereként kapott pontszámmal.

```
…
xi_g = np.empty((0, 2), float)
w_g = np.empty((0, 1), float)

for idx, point in enumerate(points):
    for idx2, point2 in enumerate(points):
        xi_g = np.append(xi_g, [[point2, point]], axis=0)
        w_g = np.append(w_g, [[weights[idx] * weights[idx2]]], axis=0)
…
```

A következő lépésben kétdimenziós pontokat készítek a *„points”* változóban tárolt egydimenziós pontokból. Két üres NumPy tömböt hozok létre. A *(0, 2)* alakú *„xi_g”* nevű tömbben a pontok és a *(0, 1)* alakú *„w_g”* nevű tömbben pedig a súlyok lesznek eltárolva. Majd egymásba ágyazott *„for”* ciklusokkal a 3.2-es fejezetben leírt logika szerint feltöltöttem a NumPy tömböket a 3.3. ábra kvadratúrapontjaival és súlyaival.

```
…
X_tri = [[0, 0], [0, 0], [1, 0], [0, 1]]
xi_t = []
dL = [[], []]

for point in xi_g:
    xi_t.append(
        np.matmul(
            [
                (1 - point[0]) * (1 - point[1]) / 4,
                (1 + point[0]) * (1 - point[1]) / 4,
                (1 + point[0]) * (1 + point[1]) / 4,
                (1 - point[0]) * (1 + point[1]) / 4,
            ],
            X_tri,
        )
    )
    dL[0].append(
        [
            -(1 - point[1]) / 4,
            (1 - point[1]) / 4,
            (1 + point[1]) / 4,
            -(1 + point[1]) / 4,
        ]
    )
    dL[1].append(
        [
            -(1 - point[0]) / 4,
            -(1 + point[0]) / 4,
            (1 + point[0]) / 4,
            (1 - point[0]) / 4,
        ]
    )
…
```

Ha megvannak a kétdimenziós pontok, akkor ezekből háromszöget alkotunk a 3.4. ábra szerint. Az *„X_tri”* változóban a négyzet koordinátáit tároljuk. A korábban leírtakkal

összhangban úgy alkotunk háromszöget a négyzetből, hogy két koordinátáját megegyezővé tesszük. *„xi_t”* tömbben lesznek a pontok eltárolva, míg *„dL”* tömb a súlyok számításában segít.

```
…
dxi_t = [np.matmul(dL[0], X_tri), np.matmul(dL[1], X_tri)]
weights = []

for i in range(len(xi_g)):
    J = dxi_t[0][i][0] * dxi_t[1][i][1] - dxi_t[0][i][1] * dxi_t[1][i][0]
    weights.append((w_g[i] * J)[0])

weights = np.array(weights)
…
```

Kiszámítjuk a súlyokat a kétdimenziós háromszögön elhelyezett kvadratúrapontokhoz, így a súlyok összege meg fog egyezni a háromszög területével, vagyis $\frac{1}{2}$-del.

```
…
points = np.empty((0, 3), float)
for point in xi_t:
    points = np.append(
        points, [[1 - point[0] - point[1], point[0], point[1]]], axis=0
    )

return points, weights
```

Végül minden kétdimenziós pontból háromdimenziósat készít a fenti *„for”* ciklus a már fentebb ismertetett logikával. Ezeket a pontokat és súlyokat fogja visszaadni a függvény.

Az így megkapott pontok és súlyok használata nagyon egyszerű. Legyen *„triangle”* egy háromszög csúcspontjait leíró háromszor hármas mátrix, és *„area”* változóban tároljuk el ennek a háromszögnek a területét. Ekkor a háromszögre illesztett kvadratúrapontokat és ezekhez tartozó súlyokat kiszámíthatjuk az alábbi kódrészlet szerint.

```
points, weights = get_triangle_quadrature(order)
quadrature_points = points @ triangle
quadrature_weights = area * 2 * weights
```

A NumPy *„@”* operátora a mátrixszorzást jelenti. A második sor ekvivalens a következő sorokkal, melyek mindegyike mátrixszorzást hajt végre.

```
quadrature_points = points.dot(triangle)
quadrature_points = np.matmul(points, triangle)
```

A súlyok mindegyikét a fentebb említett logika szerint (ld.: 3.2-es fejezet) a terület kétszeresével megszorozva kapjuk meg a Gauss-kvadratúra helyes súlyait. A továbbiakban a függvény által visszaadott *„points”* és *„weights”* változókkal dolgozok.

5.4 BEM mátrixok feltöltése

A peremelem-módszer következő lépése a $\bar{\bar{G}}$ és $\bar{\bar{H}}$ mátrixok feltöltése. Az elkészült Python programban a *„get_BEM_matrices()”* függvény felelős ezért.

5.4.1 Mátrixok álló hangforrás esetén

Először vizsgáljuk meg azt, hogy hogyan lehet feltölteni a BEM mátrixokat akkor, ha álló hangforrást modellezünk.

Vizsgáljuk meg a *„get_BEM_matrices()”* függvényt ebben az esetben az előző, 5.3-as fejezetben ismertetett módszerrel.

```
def get_BEM_matrices(triangles, centerpoints, areas, normals, points, weights, k):
…
```

Vizsgáljuk meg a függvény paramétereit. A *„triangles”* a geometria háromszögeit tartalmazó *(n, 3, 3)* alakú NumPy tömb, ahol *n* a háromszögek száma (ld.: 5.1.2-es fejezet). A *„centerpoints”*, *„areas”* és *„normals” (n, 3), (n, 1)* és *(n, 3)* alakú NumPy tömbök, melyeket a *„get_triangle_data()”* függvény számolt ki (ld.: 5.2-es fejezet). A *„points”* és *„weights” (N^2, 3)* és *(N^2, 1)* alakú NumPy tömbök a *„get_triangle_quadrature()”* függvény visszatérési értékei (ld.: 5.3-as fejezet), tehát a Gauss-kvadratúra pontjait és súlyait tartalmazzák, és *N* a kvadratúra pontszámát jelöli. *„k”* paraméter a hullámszám.

```
…
number_of_triangles = len(triangles)
…
```

A *„number_of_triangles”* változóban eltárolom azt, hogy hány háromszöget tartalmaz a geometria, mert ezt később több *„for”* ciklusban is használni fogom.

```
…
triangle_quadrature_weights = (areas * 2)[:, np.newaxis] * weights
…
```

Ez a sor a 3.2-es fejezetben leírt logika szerint kiszámítja a háromszögekhez tartozó Gauss-kvadratúra súlyait. A létrejött NumPy tömb *(n, N^2)* alakú lesz, ahol *n* a háromszögek száma, *N* pedig a kvadratúra pontszáma. A *„weights”* tömb tárolja a

kvadratúra súlyait az összes háromszögre és az *„areas”* tömb az összes háromszög területét. Következésképpen a „triangle_quadrature_weights” NumPy tömb is az összes háromszöghöz tárolja el a módosított súlyokat.

Az *„areas”* NumPy tömb a következőkhöz hasonlóan tartalmazza az adatokat.

```
[0.02100548
 0.02307804
 0.01876876
 .
 .
 .
 0.01193143
 0.01246368
 0.01115221]
```

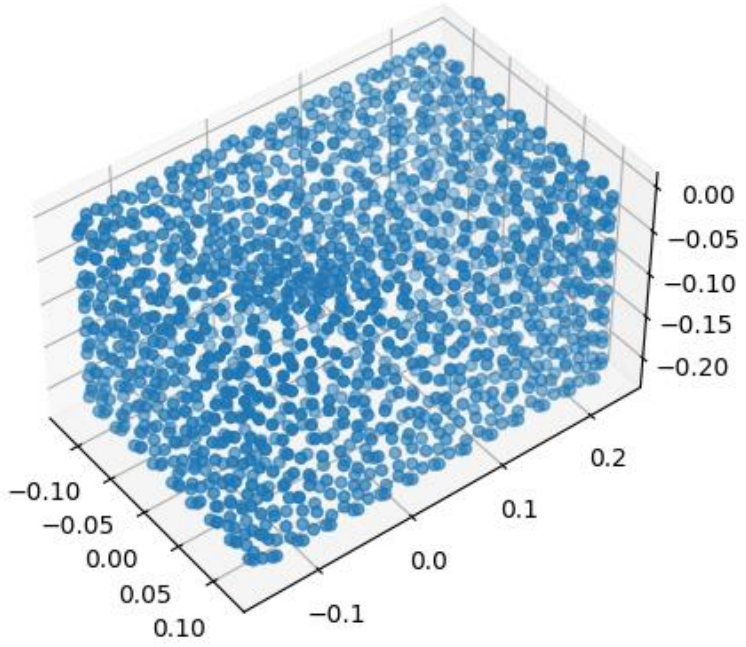
Látható, hogy mind az n darab háromszögre egy adatot tartalmaz, így a tömb alakja úgy tűnik, hogy *(n, 1).* Azonban formailag ez egy *(n,)* alakú NumPy tömb. A kódrészletben a *„[:, np.newaxis]”* rész azért felelős, hogy a tömböt ténylegesen *(n, 1)* alakúvá tegye. Az *„areas[:, np.newaxis]”* tömb az alábbi módon néz ki.

```
[[0.02100548]
 [0.02307804]
 [0.01876876]
 .
 .
 .
 [0.01193143]
 [0.01246368]
 [0.01115221]]
```

Erre azért van szükség, mert a szorzás csak így végezhető el.

```
…
quadrature_points = points @ triangles
…
```

A következő lépés a kvadratúrapontok kiszámítása szintén a 3.2-es fejezetben leírt logika szerint. Az 5.3-as fejezetben már szerepelt egy ehhez nagyon hasonló sor, viszont vegyük észre a különbséget, hogy míg ott csak egy háromszögre illesztettük a kvadratúrapontokat, itt a *„triangles”* tömb a geometria összes háromszögét tartalmazza, vagyis a létrehozott *„quadrature_points”* az összes háromszög kvadratúrapontjait tartalmazni fogja. Ennek értelmében a létrejött NumPy tömb *(n, N^2, 3)* alakú lesz. Az így kapott *„quadrature_points”* tömbre mutat egy példát az 5.4. ábra $N = 1$ értékkel a jobb láthatóság kedvéért.



5.4. ábra: Kvadratúrapontok a geometria minden háromszögén

```
…
r_vec = quadrature_points[:, :, np.newaxis] - centerpoints
…
```

Ennek a sornak a megértéséhez tegyük fel, hogy *„q_points”* egy tetszőleges háromszögre illesztett kvadratúrapontokat tartalmazó tömb, míg *„cp”* egy másik háromszög középpontja. Ekkor a *„q_points – cp”* művelet egy $(N^2, 3)$ alakú NumPy tömböt eredményez, ahol egy háromszög összes kvadratúrapontja és egy másik háromszög középpontja közötti vektorok vannak eltárolva. A fenti kódrészletben azonban a *„quadrature_points”* tömbben az összes háromszög kvadratúrapontjai, a *„centerpoints”* tömbben pedig az összes háromszög középpontjai vannak. Ebből adódóan *„r_vec”* NumPy tömbben az összes háromszög összes kvadratúrapontjából, az összes háromszög középpontjaiba mutató vektorok lesznek, tehát az alakja $(n, N^2, n, 3)$ lesz.

Mivel később olyan műveletek is szerepelni fognak, ahol több dimenziója lesz a NumPy tömböknek érdemes tudni, hogy ahelyett, hogy minden dimenziót *„:”*-tal jelölnénk a *„[:, :, np.newaxis]”* részben, a tömb összes dimenziója három ponttal is helyettesíthető. Tehát a *„quadrature_points[:, :, np.newaxis]”* kódrész ekvivalens a *„quadrature_points[… , np.newaxis]”* kódrésszel.

```
…
distances = np.linalg.norm(r_vec, axis=3)
…
```

A következő sorban a NumPy *„linalg.norm()”* függvényét használva meghatározzuk az *„r_vec”* tömbben tárolt vektorok hosszát. Ez azt jelenti, hogy minden háromszög minden kvadratúrapontjából, minden háromszög középpontjába mutató vektorok hosszát tároljuk, tehát a *„distances”* tömb *(n, N^2, n)* alakú lesz.

```
…
for i in range(number_of_triangles):
    distances[i, :, i] = 1
…
```

A fenti *„for”* ciklus azokat a távolságokat ahol a kvadratúrapontok és a középpont is ugyan azon a háromszögön van, egyre állítja. Ez azt a célt szolgálja, hogy elkerüljem a nullával való osztást, amikor a BEM mátrixok főátlóit számítom. Természetesen így értelmetlen eredményeket kapok, viszont később a kódban a 3.3.1.1-es fejezetben leírtak szerint számítom ki a $\bar{\bar{G}}$ mátrix főátlóját, és a 3.3.1.2-es fejezetben leírtak szerint a $\bar{\bar{H}}$ mátrix főátlójának összes értékét nullára állítom.

```
…
Gs = np.exp(-1j * k * distances) / (4 * np.pi * distances)
…
```

„Gs” tömbben tárolom el a Green-függvény kiértékelését a *„distances”* tömbben szereplő minden háromszög minden kvadratúrapontja és minden háromszög középpontja közötti távolságokkal. Vegyük észre, hogy ha a fentebbi *„for”* ciklusban azokat a távolságokat, ahol a középpont és kvadratúrapontok ugyan azon a háromszögön vannak nem állítom egyre, akkor ebben a sorban előfordulhatna nullával való osztás, ha az egyik kvadratúrapont a háromszög középpontjára esik. A *„Gs”* tömb tehát *(n, N^2, n)* alakú.

```
…
G = np.sum(triangle_quadrature_weights[…, np.newaxis] * Gs, axis=1).T
…
```

A 3.3-as fejezetben leírtak szerint $\bar{\bar{G}}$ mátrix értékeit úgy kaphatjuk meg, hogy a kvadratúrapontokra kiértékelt Green-függvényt megszorozzuk a pontokhoz tartozó súlyokkal, majd ezeket összegezzük. A NumPy *„sum()”* függvénye ezt teszi ebben a sorban. Így megkapjuk az *(n, n)* alakú $\bar{\bar{G}}$ mátrixot.

Azonban fontos észrevenni, hogy a mátrix főátlójában még hibás értékek szerepelnek a *„distances”* tömb módosítása miatt. A következő lépés tehát a főátló értékeinek kiszámítása lesz a 3.3.1.1-es fejezetben tárgyalt logika szerint.

```
…
for i in range(number_of_triangles):
    split_point = centerpoints[i]
    split_triangles = np.array(
        [
            [split_point, triangles[i][0], triangles[i][1]],
            [split_point, triangles[i][1], triangles[i][2]],
            [split_point, triangles[i][2], triangles[i][0]],
        ]
    )

    _, split_triangle_areas, _ = get_triangle_data(split_triangles)

    weights_split = (split_triangle_areas * 2)[:, np.newaxis] * weights
    quadrature_points_split = points @ split_triangles
    r_vec_split = quadrature_points_split[:, :, np.newaxis] - centerpoints
    distances_split = np.linalg.norm(r_vec_split, axis=3)
    Gs_split = (np.exp(-1j * k * distances_split) / (4 * np.pi *
distances_split))[…, i]

    G[i, i] = np.sum(weights_split * Gs_split)
…
```

A fenti *„for”* ciklus kiszámítja $\bar{\bar{G}}$ mátrix főátlójának értékeit. Az, hogy a mátrix főátlóját számoljuk, azt jelenti, hogy a középpont és a kvadratúrapontok is ugyan azon a háromszögön lesznek. Azért, hogy elkerüljük azt az esetet, hogy egy kvadratúrapont a háromszög középpontjára essen, és ez által nullával osszunk a Green-függvény kiértékelésénél, a háromszöget felbontjuk három darabra. Erre mutat egy példát a 3.6. ábra.

A *„split_point”* változót a háromszög középpontjával tesszük egyenlővé, tehát a létrehozott három kisebb háromszögnek ez a pont lesz az egyik csúcspontja. A *„split_triangles”* NumPy tömbben tároljuk el ezeket a háromszögeket. Az 5.2-es pontban ismertetett *„get_triangle_data()”* függvénnyel kiszámítjuk a három háromszög középpontját, területét és normálvektorát.

Python nyelvben aláhúzással szokás jelölni azokat a változókat, amelyeket nem fogunk használni. Látható, hogy a kódban a kisebb háromszögek középpontjai és normálvektorai is így vannak jelölve. A középpontok nem kellenek, hiszen az eredeti, nagyobb háromszög középpontját használjuk az integráláshoz. A normálvektorokra sem lesz szükség, mert a kisebb háromszögek normálvektorai természetesen megegyeznek a nagyobb háromszög normálvektorával. A háromszögek területeit azonban eltároltam a

„split_triangle_areas” tömbben, mivel ezeket használom a Gauss-kvadratúra súlyainak kiszámításához.

A *„for”* ciklus további részében a már fentebb ismertetett lépésekkel kiszámolom a Green-függvény integráltját a kisebb háromszögek fölött, majd ennek a három integrálnak az összege fogja megadni a $\bar{\bar{G}}$ mátrix főátlójának egy elemét.

Ezzel megkaptuk a teljes $\bar{\bar{G}}$ mátrixot feltöltve komplex értékekkel.

```
…
r_norm = r_vec / distances[…, np.newaxis]
…
```

A következő lépés az *„r_vec”* tömbben tárolt vektorok normalizálása. Ehhez minden vektort el kell osztanunk a hosszával. Az így kapott *„r_norm”* NumPy tömbben lesznek a minden háromszög minden kvadratúrapontjából, minden háromszög középpontjába mutató egység hosszú vektorok, tehát a kapott tömb *(n, N^2, n, 3)* alakú lesz.

```
…
grad_r_ny = np.sum(normals * np.transpose(r_norm, (2, 1, 0, 3)), axis=3)
…
```

Mivel $\bar{\bar{H}}$ mátrixba a Green-függvény normális irányú deriváltjának integráltjai kerülnek, ki kell számítani a vektormező gradiensét. A *„grad_r_ny”* NumPy tömb alakja *(n, N^2, n)* lesz.

```
…
Hs = (
    -(np.exp(-1j * k * distances) / (4 * np.pi * distances**2))
    * (1 + 1j * k * distances)
    * np.transpose(grad_r_ny, (2, 1, 0))
)
…
```

Így már ki tudjuk értékelni a Green-függvény normális irányú deriváltját minden háromszög minden kvadratúrapontja és minden háromszög középpontja között. Az így létrehozott *„Hs”* tömb *(n, N^2, n)* alakú lesz.

```
…
H = np.sum(triangle_quadrature_weights[…, np.newaxis] * Hs, axis=1).T
…
```

Ahogy a $\bar{\bar{G}}$ mátrixnál tettük, itt is beszorozzuk a kapott eredményeket a kvadratúrapontok súlyával, majd ezek összege adja meg az *(n, n)* alakú $\bar{\bar{H}}$ mátrixot.

Természetesen mivel a *„distances”* tömböt módosítottuk a főátlóban itt is hibás eredményeket kapunk, így ezeket még javítani kell.

```
…
np.fill_diagonal(H, 0)
…
```

A 3.3.1.2-es fejezetben leírtak szerint a $\overline{\overline{H}}$ mátrix főátlójának elemeit nullára állítjuk a NumPy *„fill_diagonal()”* függvényével. Ezzel elkészült a $\overline{\overline{H}}$ mátrix.

```
…
return G, H
```

Végül a függvény visszaadja a feltöltött $\overline{\overline{G}}$ és $\overline{\overline{H}}$ mátrixokat.

5.4.2 Mátrixok mozgó hangforrás esetén

Az 5.4.1-es fejezet *„get_BEM_matrices()”* függvényén elvégzett pár változtatással megkaphatjuk egy mozgó hangforrás $\overline{\overline{G}}$ és $\overline{\overline{H}}$ mátrixait. Annyit kell tenni, hogy a Helmholtz-egyenlet Green-függvényét (ld.: 2.3-as fejezet) és annak a normális irányú deriváltját lecseréljük az adjungált PDE Green-függvényére (ld.: 2.6.2-es fejezet) és annak a normális irányú deriváltjára.

Ehhez ki kell számítani a függvény K, R és B változóit. Vizsgáljuk meg a módosított *„get_BEM_matrices()”* függvényt.

```
def get_BEM_matrices(triangles, centerpoints, areas, normals, points,
weights, k, Mav):
…
```

A függvény kapott egy új paramétert. A *„Mav”* a test mozgását leíró *(3, 1)* alakú Mach-szám vektor.

```
…
number_of_triangles = len(triangles)
M = np.linalg.norm(Mav)
Q = 1 - M**2
Kappa = k / Q
…
```

A háromszögek száma mellett kiszámítjuk *„M”*-et, azaz a Mach-számot, ami a Mach-szám vektor hossza. *„Q”* változóban $1 - M^2$ értéket tároljuk el, és ennek segítségével számoljuk ki *„Kappa”* változót, illetve később a *„distances”* változót.

```
…
triangle_quadrature_weights = (areas * 2)[:, np.newaxis] * weights
```

```
quadrature_points = points @ triangles
r_vec = quadrature_points[:, :, np.newaxis] - centerpoints
…
```

A Gauss-kvadratúra pontjait és súlyait a háromszögeken, illetve a *„r_vec”* változót az 5.4.1-es fejezetben mutatott, álló hangforrás *„get_BEM_matrices()”* függvényével teljesen azonos módon számítjuk.

```
…
B = r_vec @ Mav.T
distances = np.sqrt(Q * np.sum(r_vec**2, axis=3) + B**2)
…
```

A *„B”* változót egy mátrixszorzással kiszámíthatjuk, míg a *„distances”* tömbbe a 2.6.2-es fejezetben leírt súlyozott távolságok kerülnek.

```
…
for i in range(number_of_triangles):
    distances[i, :, i] = 1
…
```

Ez a *„for”* ciklus szintén megegyezik az 5.4.1-es fejezetben láthatóval, és most is a nullával történő osztást hivatott megakadályozni.

```
…
Gs = np.exp(-1j * Kappa * (distances + B)) / (4 * np.pi * distances)
G = np.sum(triangle_quadrature_weights[…, np.newaxis] * Gs, axis=1).T
…
```

A $\bar{\bar{G}}$ mátrix elemeit ezúttal az adjungált PDE Green-függvényének minden kvadratúrapontra történő kiértékelésével kaphatjuk meg. Minden így kapott értéket beszorozva a ponthoz tartozó súllyal, majd ezeket összegezve kapjuk meg a $\bar{\bar{G}}$ mátrixot.

```
…
r_norm = r_vec / np.linalg.norm(r_vec, axis=3)[…, np.newaxis]
grad_r_ny = np.sum(normals * np.transpose(r_norm, (2, 1, 0, 3)), axis=3)
…
```

Ezeket a sorokat szintén kevés változtatással át tudjuk emelni az álló hangforráshoz írt *„get_BEM_matrices()”* függvényből. Egyedül arra kell figyelni, hogy az 5.4.1-es fejezetben a vektorok normalizálásához elég volt a *„distances”* tömbbel osztani, hogy megkapjuk az egység hosszú normálvektorokat, azonban most súlyozott távolságokat tárolunk ebben a változóban, így a vektorok hosszát ki kell számítanunk az osztáshoz.

```
…
Hs = (
    -(np.exp(-1j * Kappa * (distances + B)) / (4 * np.pi * distances**2))
    * (1 + 1j * Kappa * distances)
    * np.transpose(grad_r_ny, (2, 1, 0))
)

H = np.sum(triangle_quadrature_weights[…, np.newaxis] * Hs, axis=1).T
…
```

A $\bar{\bar{G}}$ mátrixhoz hasonlóan a $\bar{\bar{H}}$ mátrixnál is az adjungált PDE Green-függvényével kell dolgoznunk. A 2.3-as fejezet Green-függvényének normális irányú deriváltja helyett a 2.6.2-es fejezet Green-függvényének normális irányú deriváltját kell használnunk.

```
…
for i in range(number_of_triangles):
    split_point = centerpoints[i]
    split_triangles = np.array(
        [
            [split_point, triangles[i][0], triangles[i][1]],
            [split_point, triangles[i][1], triangles[i][2]],
            [split_point, triangles[i][2], triangles[i][0]],
        ]
    )

    _, split_triangle_areas, split_triangle_normals = get_triangle_data(
        split_triangles
    )

    weights_split = (split_triangle_areas * 2)[:, np.newaxis] * weights
    quadrature_points_split = points @ split_triangles
    r_vec_split = quadrature_points_split[:, :, np.newaxis] - centerpoints
    B_split = r_vec_split @ Mav.T
    distances_split = np.sqrt(Q*np.sum(r_vec_split**2, axis=3) + B_split**2)
    r_norm_split = (
        r_vec_split / np.linalg.norm(r_vec_split, axis=3)[…, np.newaxis]
    )
    grad_r_ny_split = np.sum(
        split_triangle_normals*np.transpose(r_norm_split,(2,1,0,3)), axis=3
    )

    Gs_split = (
        np.exp(-1j * Kappa * (distances_split + B_split))
        / (4 * np.pi * distances_split)
    )[…, i]
    Hs_split = (
        -(
            np.exp(-1j * Kappa * (distances_split + B_split))
            / (4 * np.pi * distances_split**2)
        )
        * (1 + 1j * Kappa * distances_split)
        * np.transpose(grad_r_ny_split, (2, 1, 0))
    )[…, i]
```

```
        G[i, i] = np.sum(weights_split * Gs_split)
        H[i, i] = np.sum(weights_split * Hs_split)
    …
```

Ez a *„for”* ciklus is majdnem megegyezik az álló hangforrás függvényénél látottal, viszont fontos különbség, hogy itt nem csak a $\bar{\bar{G}}$ mátrix, de $\bar{\bar{H}}$ mátrix főátlójának elemeit is ki kell számolnunk, a 3.3.1.2-es fejezet értelmében.

```
    …
    return G, H
```

Természetesen a függvény itt is visszaadja a feltöltött $\bar{\bar{G}}$ és $\bar{\bar{H}}$ mátrixokat.

6 Transzparens teszt

A transzparens tesztet vagy átlátszó tesztet azért hívjuk így, mert a geometriától független. Ennek segítségével ellenőrizhetjük a $\bar{\bar{G}}$ és $\bar{\bar{H}}$ mátrixokat.

6.1.1 Transzparens teszt álló hangforráshoz

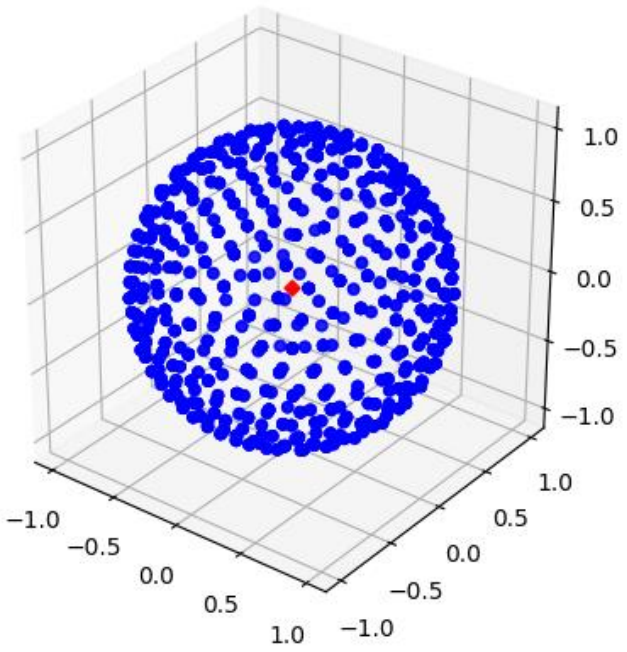
Kezdem most is azzal, hogy bemutatom, hogy egy álló hangforrás esetében, hogyan működik a transzparens teszt, és ebből kiindulva áttérek a mozgó hangforrás esetére.

```
def transparent_test(triangles, centerpoints, normals, k, G, H) -> float:
…
```

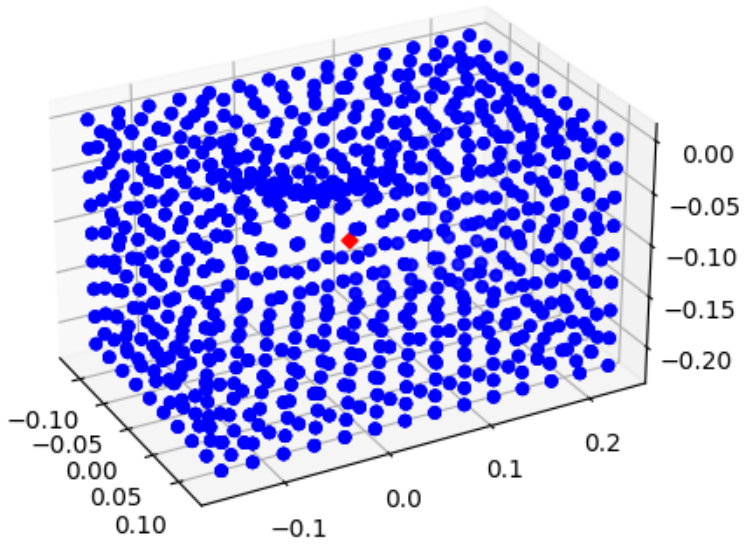
A függvény paramétere a geometria háromszögeit tartalmazó *„triangles”* tömb, az ezeknek a háromszögeknek a középpontjait tartalmazó *„centerpoints”* tömb és a háromszögekhez tartozó normálvektorokat tartalmazó *„normals”* tömb. A *„k”* paraméter a hullámszámot jelöli, és további paraméterek a feltöltött $\bar{\bar{G}}$ és $\bar{\bar{H}}$ mátrixok. *„→float”* azt jelenti, hogy a függvény egy valós számmal tér vissza.

```
…
xs, ys, zs = triangles.transpose((2, 0, 1)).reshape(3, -1)
point = np.mean(np.array([xs, ys, zs]), axis=1)
…
```

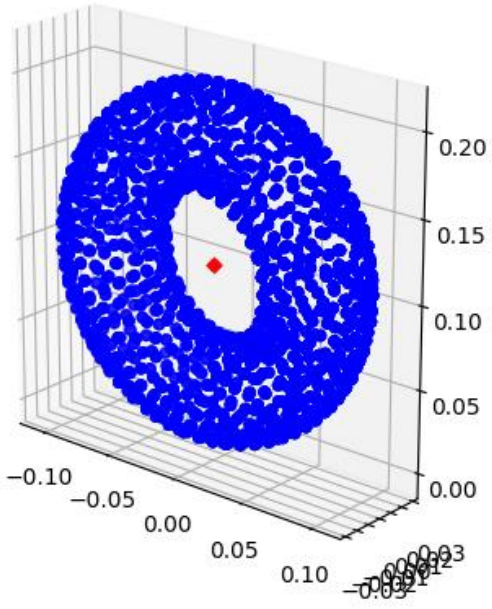
A transzparens teszt első lépése az, hogy létre kell hoznunk egy pontot a geometria belsejében. Ez a két sor ezért felelős. A *„point”* változóban egy olyan pontot fogunk eltárolni, ami az összes háromszög, összes csúcspontjának átlaga. Egy ilyen pontot mutat az 6.1. ábra, ahol kék pontok jelölik a háromszögek csúcspontjait, és piros gyémánt ezeknek a pontoknak az átlagát. Ennél a geometriánál nem nehéz belátni, hogy a pont valóban a geometria belsejébe esik, sőt az nagyjából a gömb középpontja lesz. Fontos megjegyezni azonban, hogy ez a logika azonban nem feltétlenül biztosítja, hogy a *„point”* változóban eltárolt pont a geometria belsejében legyen. Az általam használt hangszóró geometrián ez az algoritmus a test belsejében hoz létre egy pontot. Ezt mutatja be az 6.2. ábra. Azonban nem nehéz egy olyan testet elképzelni, ahol a csúcspontok átlaga által meghatározott pont kívül esik a geometrián. Erre mutat egy példát az 6.3. ábra, ahol a meghatározott pont a geometrián kívül esik. Tehát ez a logika a bemutatott geometriákon működik, viszont elképzelhető olyan geometria, hogy a testen belüli pontot manuálisan kell elhelyezni.



6.1. ábra: Csúcspontok átlaga egy egységgömb geometrián



6.2. ábra: Csúcspontok átlaga a hangszóró geometrián



6.3. ábra: Csúcspontok átlaga egy fánk geometrián

```
…
r_vec = centerpoints - point
distances = np.linalg.norm(r_vec, axis=1)[…, np.newaxis]
…
```

Az 5.4.1-es fejezetben bemutatott kóddal megegyező módon kiszámítom az *„r_vec”* és a *„distances”* változók értékeit a háromszögek középpontjai és a létrehozott pont között.

```
…
pa = np.exp(-1j * k * distances) / (4 * np.pi * distances)
…
```

A *„pa”* változóba kiértékeljük a Green-függvényt (ld.: 2.3-as fejezet) a középpontok es a csúcspontok átlagaként meghatározott pont között.

```
…
r_norm = r_vec / distances
grad_r_ny = np.sum(normals * r_norm, axis=1)[…, np.newaxis]

q = (
    -(np.exp(-1j * k * distances) / (4 * np.pi * distances**2))
    * (1 + 1j * k * distances)
    * grad_r_ny
)
…
```

Normalizálom a vektorokat és kiszámítom a *„grad_r_ny”* változót, majd *„q”* vektorba kiértékelem a Green-függvény normális irányú deriváltját a háromszögek középpontjai és a *„point”* változó pontja között.

```
…
A = H - np.eye(len(triangles)) / 2
b = G @ q
…
```

Ez a két sor „A” mátrix és „b” vektor kiszámítását végzik el a következők szerint:

$$\bar{\bar{A}} = \bar{\bar{H}} - \frac{1}{2}\bar{\bar{I}},$$

$$\bar{b} = \bar{\bar{G}} \cdot \bar{q}.$$

$\bar{\bar{I}}$ az egységmátrix. Ezekre a változókra azért van szükség, mert $\bar{p}$ kiszámítható a következőképpen:

$$\bar{p} = \bar{\bar{A}}^{-1} \cdot \bar{b}.$$

```
…
p = np.linalg.solve(A, b)
…
```

$\bar{\bar{A}}^{-1}$ az $\bar{\bar{A}}$ mátrix inverzét jelenti, aminek a kiszámítása nagyon erőforrásigényes, így a lineáris egyenletrendszer megoldását a NumPy *„linalg.solve()”* függvénye végzi el. A most kiszámított *„p”* vektor meg kell egyezzen az analitikusan kiszámított *„pa”* vektorral.

```
…
return np.linalg.norm(pa - p) / np.linalg.norm(pa)
```

A függvény visszatérési értéke a relatív hiba lesz *„p”* és *„pa”* vektorok között.

6.1.2 Transzparens teszt mozgó hangforráshoz

Kevés változtatással módosíthatjuk úgy az előző pont függvényét, hogy mozgó hangforrásra alkalmazható legyen.

```
def transparent_test(triangles, centerpoints, normals, k, G, H, Mav):
…
```

A függvény a 6.1.1-es fejezet paraméterei mellett, kap még egy *„Mav”* paramétert, ami az 5.4.2-es fejezett megegyező névű paraméterével azonos Mach-szám vektor.

```
…
M = np.linalg.norm(Mav)
Q = 1 - M**2
Kappa = k / Q
…
```

A következő sorok szintén megegyeznek az 5.4.2-es fejezet *„get_BEM_matrices()”* függvényében találhatókkal. Kiszámítom a Mach-számot, *„Kappa”*-t és a távolságok számításához használt *„Q”* változót.

```
…
xs, ys, zs = triangles.transpose((2, 0, 1)).reshape(3, -1)
point = np.mean(np.array([xs, ys, zs]), axis=1)
…
```

A 6.1.1-es fejezettel megegyező módon, most is veszek egy pontot a geometria belsejében, tehát a háromszögek csúcspontjainak átlagát.

```
…
r_vec = centerpoints - point
B = r_vec @ Mav.T
distances = np.sqrt(Q * np.sum(r_vec**2, axis=1) + B**2)[..., np.newaxis]
…
```

Ugyanúgy, mint az álló hangforrás transzparens tesztjében itt is kiszámolom az *„r_vec”* és a *„distances”* változókat. A különbség az, hogy itt a távolságok számításához nem elég az *„r_vec”* vektorok hosszait venni, hanem a 2.6.2-es fejezetben leírtak szerint kell számolni, és ehhez a *„B”* változót is kiszámítom.

```
…
pa = np.exp(-1j*Kappa*(distances-B[…, np.newaxis])) / (4 * np.pi * distances)
…
```

A 6.1.1-es fejezethez hasonlóan *„pa”* változót itt is a Green-függvény kiértékelésével minden távolságra kaphatjuk meg. Azonban figyelni kell, hogy ezúttal nem az álló hangforrás (ld.: 2.3-as fejezet), hanem a mozgó hangforrás Green-függvényét (ld.: 2.6.2-es fejezet) használjuk. Vegyük észre, hogy a fenti sorban *„B”* előjelét megváltoztattam. Erre a 2.6-os fejezetben leírtak miatt van szükség, tehát meg kell változtatni „Mav” vektor irányát. Ennek hatására *„B”* előjele meg fog változni, azonban a távolságok számításakor *„B”* négyzetét vesszük, így itt nem kell változtatni az értékeken.

```
…
r_norm = r_vec / np.linalg.norm(r_vec, axis=1)[..., np.newaxis]
grad_r_ny = np.sum(normals * r_norm, axis=1)[..., np.newaxis]
…
```

A *„r_norm”* és *„grad_r_ny”* változok számítása a 6.1.1-es fejezettel azonos módon történik, azzal a kivétellel, hogy ezúttal nem elég az *„r_vec”* változót a *„distances”* változóval elosztani a vektorok normalizálásához, hiszen ebben most nem a vektorok hosszai vannak.

```
…
q = (
    -(np.exp(-1j * Kappa * (distances + B[…, np.newaxis]))
    / (4 * np.pi * distances**2))
    * (1 + 1j * Kappa * distances)
    * grad_r_ny
)
…
```

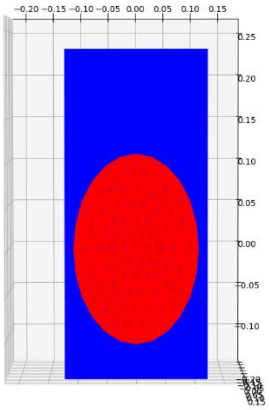
Az álló hangforrás esetéhez hasonlóan, ahol a *„q”* változóba ki értékelem a Green-függvény normális irányú deriváltját, most is ezt teszem, csak a 2.6.2-es fejezet Green-függvényét használva.

```
…
A = H - np.eye(len(triangles)) / 2
b = G @ q

p = np.linalg.solve(A, b)

return np.linalg.norm(pa - p) / np.linalg.norm(pa)
```

A 6.1.1-es fejezetben látható *„transparent_test()”* függvénnyel teljesen megegyező módon oldom meg a lineáris egyenletrendszert, és adom vissza a megoldás hibáját. Mozgó hangforrás esetében minél nagyobb Mach-számot használunk, annál pontatlanabbak lesznek a közelítéseink.

7 Sugárzás modellezése

Miután kiszámoltuk a BEM mátrixokat a geometria felületén a lesugárzott hang egyszerűen modellezhető. A továbbiakban a már korábban bemutatott hangszóró geometriát (ld.: 5.2. ábra) fogom használni. Az első lépés a hangszóró membránján egy gerjesztést megadni.



7.1. ábra: A hangszóró membránja

A 7.1. ábra hangszóróján pirossal jelölt részén, azaz a hangszóró membránján a gerjesztést százra, míg a hangszóró többi részén nullára állítom.

```
conditions = [
    np.all(triangles[:, :, 0] == -0.125, axis=1),
    np.all(triangles[:, :, 0] == 0.125, axis=1),
    np.all(triangles[:, :, 1] == 0.23, axis=1),
    np.all(triangles[:, :, 1] == -0.15, axis=1),
    np.all(triangles[:, :, 2] == -0.22, axis=1),
    np.all(triangles[:, :, 2] == 0, axis=1),
]

condition = np.logical_or.reduce(conditions)
q = np.where(condition, 0, 100)[..., np.newaxis]
…
```

Ebben a kódrészletben feltételeket határozok meg minden olyan háromszögre, amelynek az összes csúcspontja a hangszóró valamelyik lapjának síkjára esik. Majd ezeknek a feltételeknek a logikai vagy művelettel történő összekapcsolásával kapok egy feltételt

azokra a háromszögekre, amelyek a hangszóró házán helyezkednek el. A *„q”* változóba nulla kerül ott, ahol a feltétel igaz, és száz, ahol a feltétel hamis.

```
…
A = H - np.eye(len(triangles)) / 2
b = G @ q
p = np.linalg.solve(A, b)
…
```

Ha ismert *„q”* a felületen, és ismertek a BEM mátrixok akkor ki tudjuk számolni *„p”*-t a felületen, ugyanúgy, ahogy a transzparens tesztnél (ld.: 6-os fejezet).

```
…
xs, ys, zs = triangles.transpose((2, 0, 1)).reshape(3, -1)
cp = np.mean(np.array([xs, ys, zs]), axis=1)
radius = 1
num_points = 360
theta = np.linspace(0, 2*np.pi, num_points)
x = cp[0] + radius * np.cos(theta)
z = cp[2] + radius * np.sin(theta)

field_points = np.column_stack((x, cp[1] * np.ones(num_points), z))
…
```

Szintén a transzparens tesztnél látott módon meghatározzuk a geometria középpontját. Ez lesz a középpontja annak a körnek, aminek a mentén elhelyezzük a pontokat, ahol majd mérjük a nyomást. A fenti kód egy méter sugarú kör mentén egyenletesen elhelyez háromszázhatvan pontot.

```
…
Gf, Hf = get_field_BEM_matrices(triangles, field_points, areas, normals,
points, weights, k, Mav)
…
```

A következő lépés a $\bar{\bar{G}}_f$ és a $\bar{\bar{H}}_f$ mátrixok kiszámítása. A *„get_field_BEM_matrices()”* függvény szinte teljesen megegyezik az 5.4.2-es fejezetben látható *„get_BEM_matrices()”* függvénnyel. A különbség annyi, hogy a függvény második paramétere nem a háromszögek középpontjai, hanem az előbb meghatározott kör menti pontok, azaz a *„field_points”* NumPy tömb. Ennek az lesz a következménye, hogy a pont nem fog a háromszög belsejébe esni, így nem kell speciális esetként kezelni a mátrixok átlóinak számítását. Ennek köszönhetően a függvény a következőképpen egyszerűsíthető.

```
def get_field_BEM_matrices(triangles, field_points, areas, normals, points,
weights, k, Mav):
    M = np.linalg.norm(Mav)
    Q = 1 - M**2
    Kappa = k / Q
```

```
    triangle_quadrature_weights = (areas * 2)[:, np.newaxis] * weights
    quadrature_points = points @ triangles
    r_vec = quadrature_points[:, :, np.newaxis] - field_points
    B = r_vec @ Mav.T
    distances = np.sqrt(Q * np.sum(r_vec**2, axis=3) + B**2)

    Gs = np.exp(-1j * Kappa * (distances + B)) / (4 * np.pi * distances)

    Gf = np.sum(triangle_quadrature_weights[..., np.newaxis] * Gs, axis=1).T

    r_norm = r_vec / np.linalg.norm(r_vec, axis=3)[..., np.newaxis]
    grad_r_ny = np.sum(normals * np.transpose(r_norm, (2, 1, 0, 3)), axis=3)

    Hs = (
        -(np.exp(-1j * Kappa * (distances + B)) / (4 * np.pi * distances**2))
        * (1 + 1j * Kappa * distances)
        * np.transpose(grad_r_ny, (2, 1, 0))
    )

    Hf = np.sum(triangle_quadrature_weights[..., np.newaxis] * Hs, axis=1).T

    return Gf, Hf
```

Ha a mátrixok ismertek, akkor a nyomás a meghatározott pontokban a következő képlettel számítható ki:

$$\bar{p}_f = \bar{\bar{H}}_f \cdot \bar{p} - \bar{\bar{G}}_f \cdot \bar{q}.$$

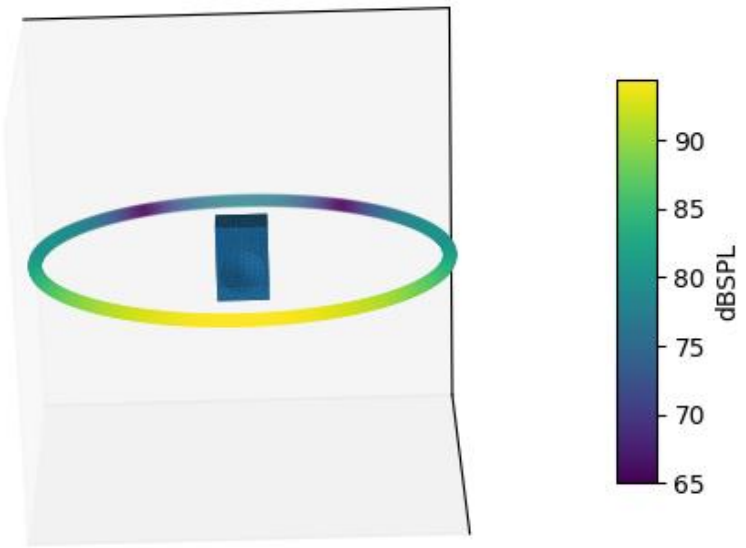
Tehát Pythonban:

```
pf = Hf @ p - Gf @ q
```

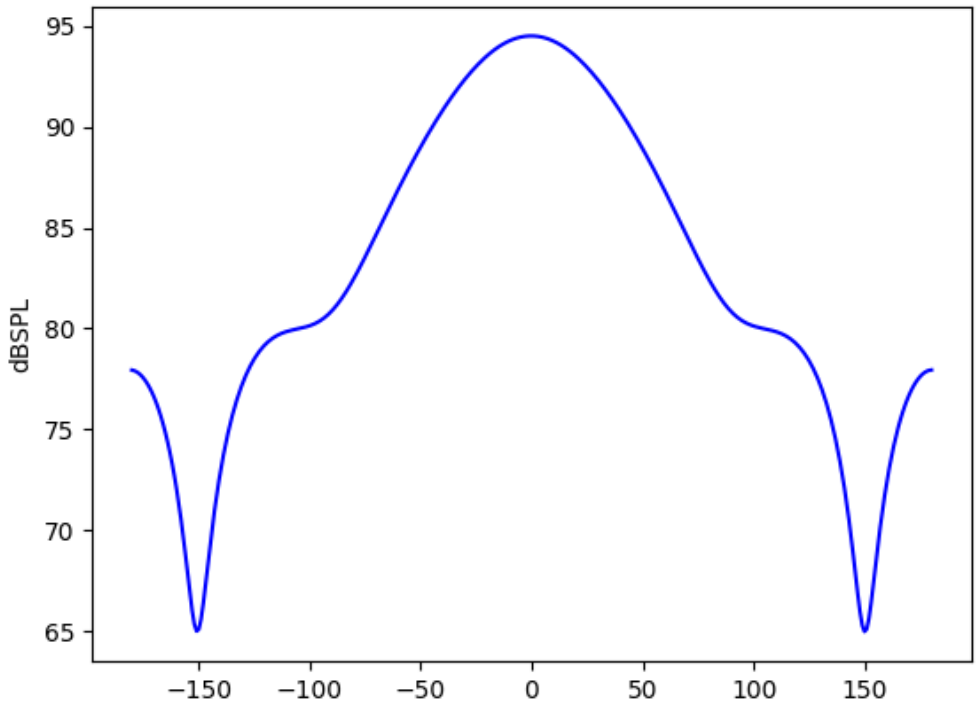
A $\bar{p}_f$ vektorban a pontokra számított nyomásértékek vannak. Ebben az esetben háromszázhatvan darab komplex szám. A nyomásokat pascalban kapjuk meg, ha a komplex számok abszolútértékeit vesszük. A megjelenítésénél, a jobb láthatóság érdekében, én az értékeket átváltottam decibelre. Ezt a következőképpen lehet megtenni:

```
pf = 20 * np.log10(np.abs(pf) / 0.2e-4)
```

A pontokra számított nyomásokat 1000 Hz frekvencián szemlélteti a 7.2. ábra. Mivel így nem lehet az eredményeket rendesen átlátni a továbbiakban a kör pontjait $-180°$-tól $180°$-ig ábrázolom, így a hangszóró $0°$ felé fog nézni. Erre mutat egy példát a 7.3. ábra. Ebben a formában sokkal könnyebben le lehet olvasni a nyomás értékét egyes helyeken.



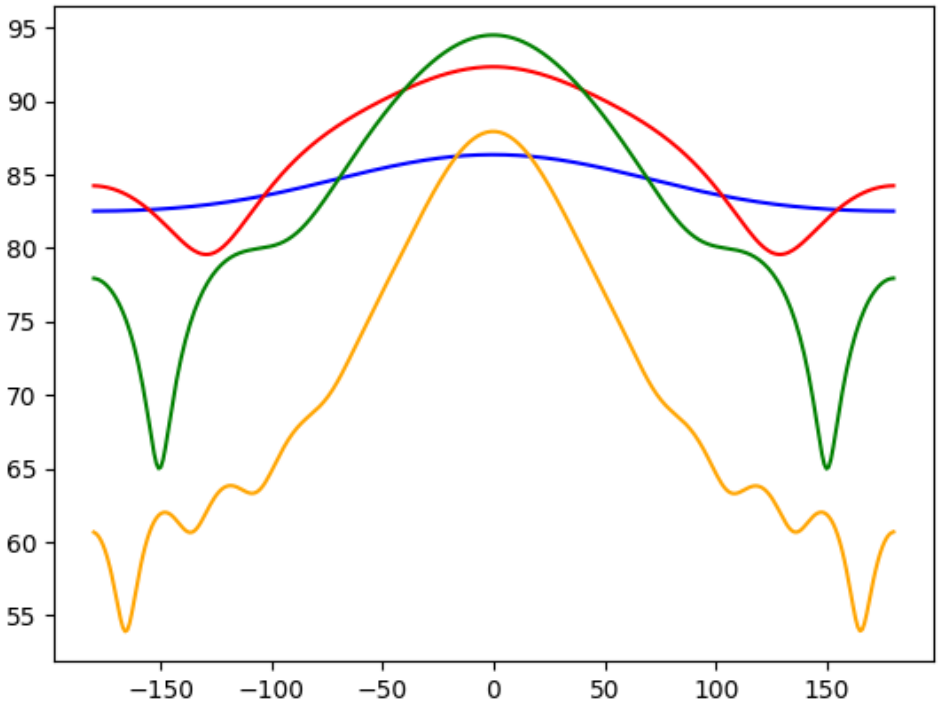
7.2. ábra: Pontok a hangszóró kőrül



7.3. ábra: Álló hangszóró iránykarakterisztikája a peremelem-módszerrel 1000 Hz frekvencián, $-180°$-tól $180°$-ig ábrázolva

8 Eredmények értékelése

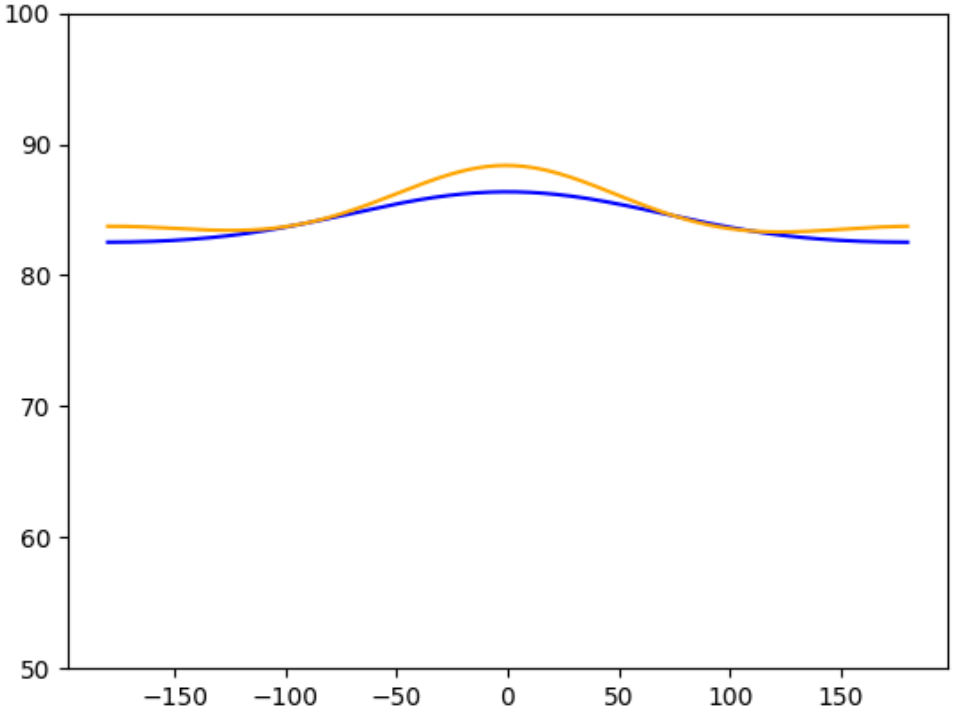
Először vizsgáljuk meg álló esetben a hangszóró iránykarakterisztikáját. A fejezet grafikonjain az y tengely minden esetben az amplitúdó lesz dBSPL-ben, míg az x tengely a körön felvett pontok fokban.



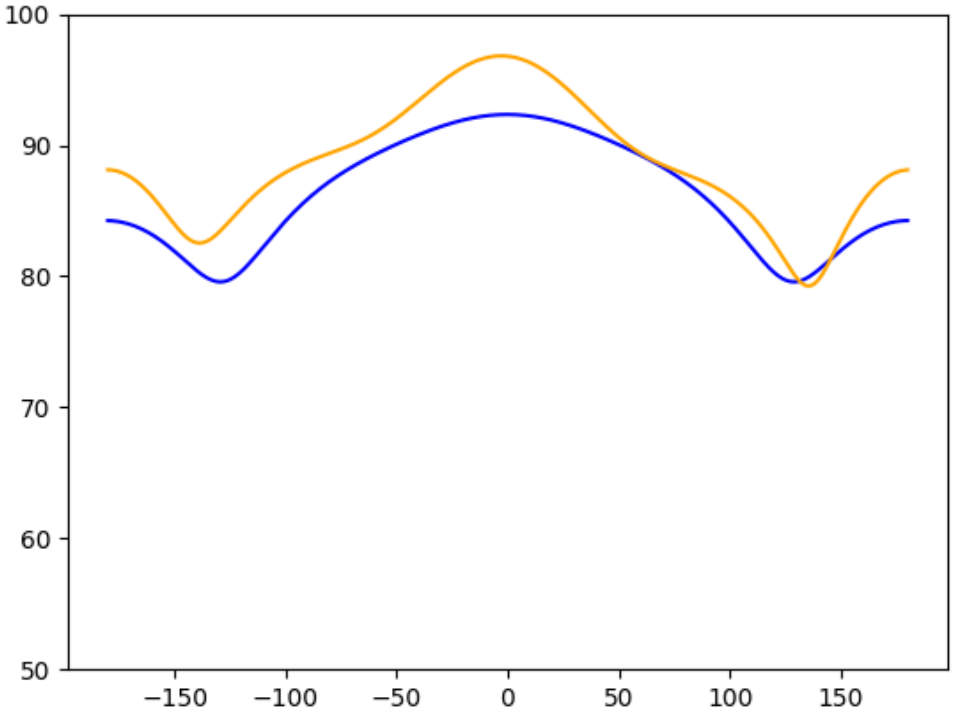
8.1. ábra: Nyomások különböző frekvenciákon

A pontokra számított nyomásokat mutatja a 8.1. ábra különböző frekvenciákon. Kék színnel van jelölve 100 Hz, pirossal 500 Hz, zölddel 1000 Hz és narancssárgával 2000 Hz. Azt vehetjük észre, hogy alacsony frekvenciák esetén a hangszóró majdnem pontforrásként viselkedik, azaz minden ponton közel azonos a nyomás. Minél magasabb lesz a frekvencia, a hang annál inkább irányított lesz.

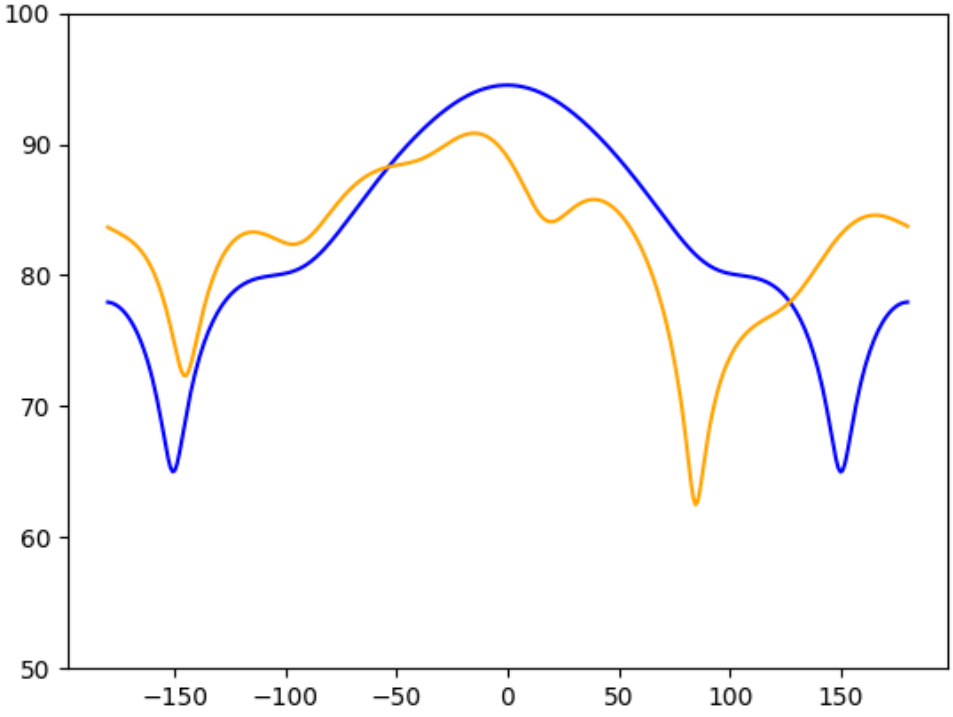
Vizsgáljuk meg, hogy mi történik a hangszóró oldal irányú mozgatása, avagy a közeg oldal irányú áramlása esetén.



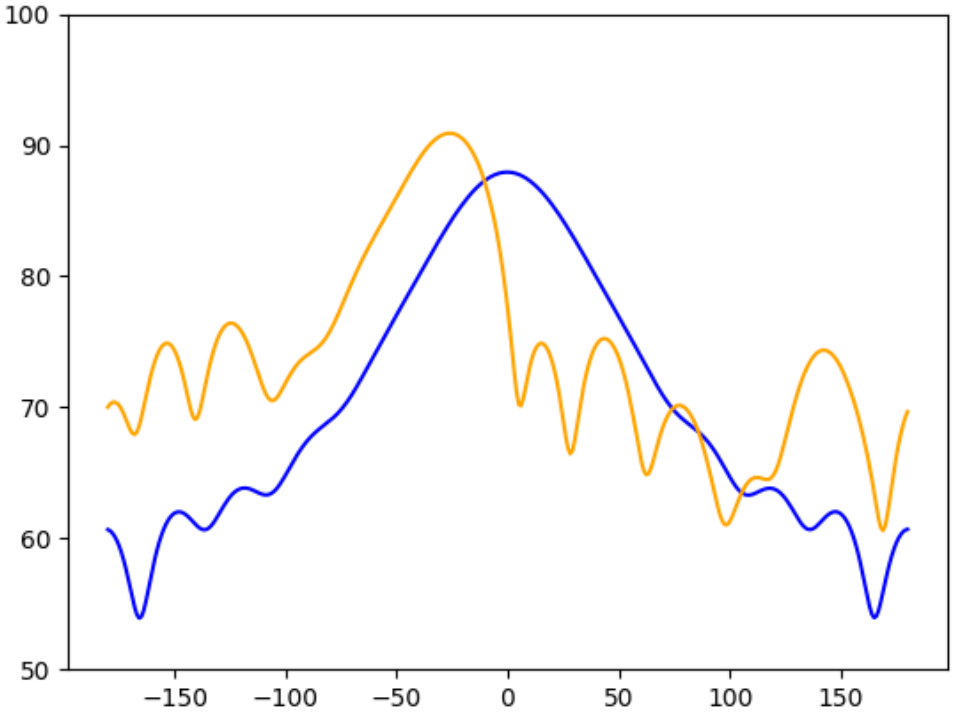
8.2. ábra: Nyomások 100 Hz frekvenciájú hang esetén



8.3. ábra: Nyomások 500 Hz frekvenciájú hang esetén

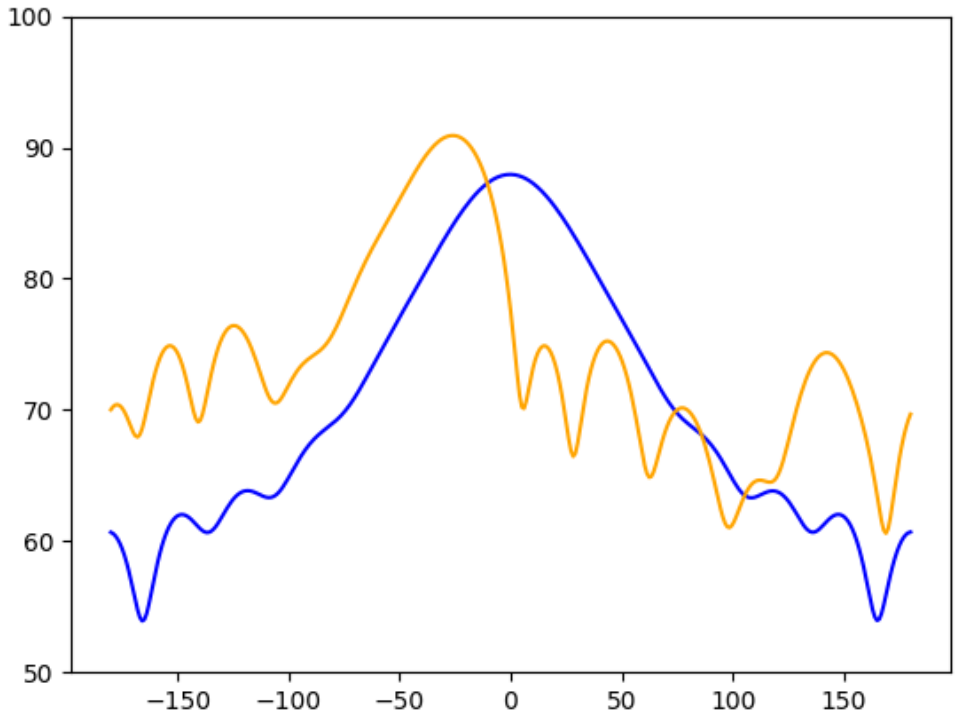


8.4. ábra: Nyomások 1000 Hz frekvenciájú hang esetén

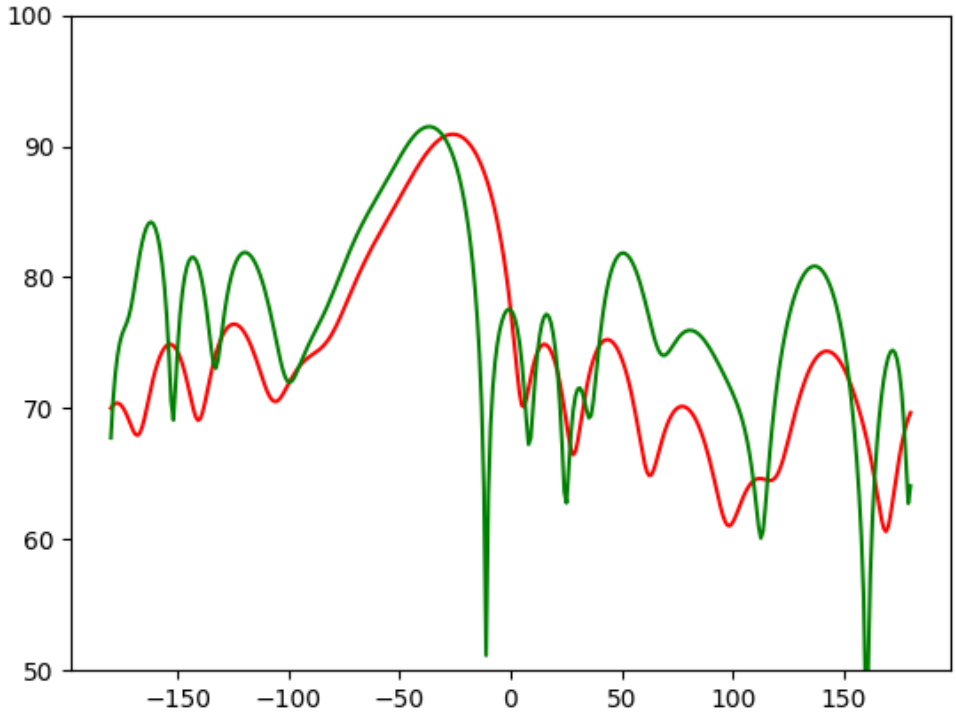


8.5. ábra: Nyomások 2000 Hz frekvenciájú hang esetén

A 8.2. ábra, a 8.3. ábra, a 8.4. ábra és 8.5. ábra mindegyike két esetet mutat 100 Hz, 500 Hz, 1000 Hz illetve 2000 Hz frekvenciájú hanghullámoknál. Kék szín jelöli a nyomásokat álló hangforrás, és narancssárga szín Mach 0,5-tel, azaz a hangsebesség felével mozgó hangforrás esetén. Azt vehetjük észre, hogy a frekvencia növelésével egyre nagyobb hatása lesz a mozgásnak a hang terjedésére.



8.6. ábra: Nyomások Mach 0 (kék) és Mach 0, 2 (narancssárga) esetén



8.7. ábra: Nyomások Mach 0, 5 (piros) és Mach 0, 7 (zöld) esetén

A 8.6. ábra és a 8.7. ábra egy 2000 Hz frekvenciájú hang esetén mutatja a nyomásokat a hangszóró körül felvett pontokon. A mérések során a Mach-számot változtattam. A kék vonal jelöli a nyomások értékeit Mach 0-nál, azaz álló hangforrásnál. A narancssárga vonal esetén a Mach-szám 0,2, piros vonal esetén 0,5, míg zöld vonal esetén 0,7 volt. Azt figyelhetjük meg, hogy a Mach-szám növelésével egyre jobban torzul a lesugárzott nyomás képe. Ez a viselkedés megfelel az elvárásainknak, hiszen a forrás haladása (vagy a közeg áramlása) főként a mozgás irányában torzítja a hullámalakokat.

9 Potenciális fejlesztések

Jelenleg az elkészült Python program legnagyobb hibájának a memóriaigényességét tartom. Ennek az 5.4-es fejezetben látható *„get_BEM_matrices()”* függvény implementálása az oka. Ennek a feladata, hogy kiértékelje a Green-függvényt és a Green-függvény normális irányú deriváltját a geometria minden háromszögének középpontja és minden háromszögre illesztett kvadratúrapontok között, majd ezeknek, a súlyokkal beszorzott összegével, azaz a háromszögek fölötti integráltakkal feltöltse a $\bar{\bar{G}}$ és a $\bar{\bar{H}}$ mátrixokat. Talán a legkézenfekvőbb megoldás a problémára a következő kód.

```
G = np.empty((number_of_triangles, number_of_triangles), np.complex_)
H = np.empty((number_of_triangles, number_of_triangles), np.complex_)

for idx1 in range(number_of_triangles):
    for idx2 in range(number_of_triangles):
        if idx1 == idx2:
            split_point = centerpoints[idx1]
            split_triangles = np.array(
                [
                    [split_point, triangles[idx2][0], triangles[idx2][1]],
                    [split_point, triangles[idx2][1], triangles[idx2][2]],
                    [split_point, triangles[idx2][2], triangles[idx2][0]],
                ]
            )

            (
                _,
                split_triangle_areas,
                split_triangle_normals,
            ) = get_triangle_data(split_triangles)

            Gs = np.empty((3, 1), np.complex_)
            for split_idx in range(3):
                Gs[split_idx], _ = integrate(
                    split_triangles[split_idx],
                    split_triangle_areas[split_idx],
                    split_triangle_normals[split_idx],
                    points,
                    weights,
                    centerpoints[idx1],
                    k
                )

            G[idx1, idx2] = Gs.sum()
            H[idx1, idx2] = 0

        else:
            g, h = integrate(
                triangles[idx2],
                areas[idx2],
                normals[idx2],
```

```
            points,
            weights,
            centerpoints[idx1],
            k
        )

        G[idx1, idx2] = g
        H[idx1, idx2] = h
```

Ebben a kódrészben létrehozok egy *„G”* és egy *„H” (n, n)* alakú üres mátrixot, majd két *„for”* ciklussal végig iterálok a háromszögeken, és minden háromszög párra meghívom az integrálást elvégző *„integrate()”* függvényt.

A ciklusban lévő elágazás azt vizsgálja, hogy a mátrixok átlóit számítjuk-e. Tehát ha *„idx1 == idx2”* igaz, az azt jelenti, hogy a mátrix főátlójának egy elemét vizsgáljuk. Ilyen esetben a 3.3.1-es fejezetben tárgyaltak szerint a háromszöget felbontom három kisebb háromszögre, és ezek mindegyikére elvégzem az integrálást. Az így kapott három eredmény összege lesz a háromszög fölötti integrál eredménye. Vegyük észre, hogy a kódban $\bar{\bar{G}}$ mátrixba ez az összeg kerül be, viszont $\bar{\bar{H}}$ mátrixba minden esetben 0. Ez azt jelenti, hogy ez az algoritmus álló hangforrás esetén fog csak működni, hiszen a $\bar{\bar{H}}$ mátrix főátlójának elemei ebben az esetben lesznek nullák.

Ha nem a mátrixok átlóit számítjuk, azaz az *„else”* ágban vagyunk akkor a mátrixokba egyszerűen beírható az *„integrate()”* függvény visszatérési értékei.

Vizsgáljuk meg az *„integrate()”* függvényt.

```
def integrate(triangle, area, normal, points, weights, point, k):
    weights = area * 2 * weights
    quadrature_points = points @ triangle
    r_vec = quadrature_points - point
    distances = np.linalg.norm(r_vec, axis=1)
    r_norm = r_vec / distances[..., np.newaxis]
    grad_r_ny = np.sum(normal * r_norm, axis=1)

    Gs = np.exp(-1j * k * distances) / (4 * np.pi * distances)
    Hs = (
        -(np.exp(-1j * k * distances) / (4 * np.pi * distances**2))
        * (1 + 1j * k * distances)
        * grad_r_ny
    )

    return np.sum(weights * Gs), np.sum(weights * Hs)
```

Nézzük meg a paramétereket. A *„triangles”* egy háromszög, amire a *„points”* paraméter kvadratúrapontjait illesztjük. A *„weights”* paraméter a kvadratúrapontokhoz tartozó súlyok. „area” és „normal” paraméterek a háromszög területe és normálvektora. A *„point”* paraméterben egy másik háromszög középpontja lesz, *„k”* pedig a hullámszám.

A függvény logikája megegyezik az 5.4-es fejezetben találhatóval, azzal a különbséggel, hogy itt csak egyetlen háromszög fölött végezzük el az integrálást. Ennek megfelelően a függvény visszatérési értékei a Green-függvény és a Green-függvény normális irányú deriváltjának integráltjai lesznek, azaz két komplex szám.

Ha ezzel az algoritmussal töltjük fel a BEM mátrixokat, akkor két *(n, n)* alakú mátrix jön létre, ahol minden elem *„np.complex_”* típusú lesz, ami azt jelenti, hogy minden elem tizenhat bájton lesz eltárolva. Ebből kiszámolhatjuk, hogy a két mátrix összesen $2 \cdot n^2 \cdot 16$ bájt memóriát igényel.

Nézzünk egy példát. Az egyik általam használt geometria 1472 háromszöget tartalmaz. Így a két mátrix memóriaigénye:

$$2 \cdot 1472^2 \cdot 16 = 69\,337\,088,$$

azaz 66,125 megabájt. Könnyen belátható, hogy a háromszögek számának növelésével, ez a memóriaigény négyzetesen nő.

Ennek a két mátrixnak az eltárolása elkerülhetetlen, viszont vegyük észre, hogy a mátrixok számítása során sokkal több memóriát nem használ ez az algoritmus, így a memóriafelhasználás szempontjából ez a leghatékonyabb.

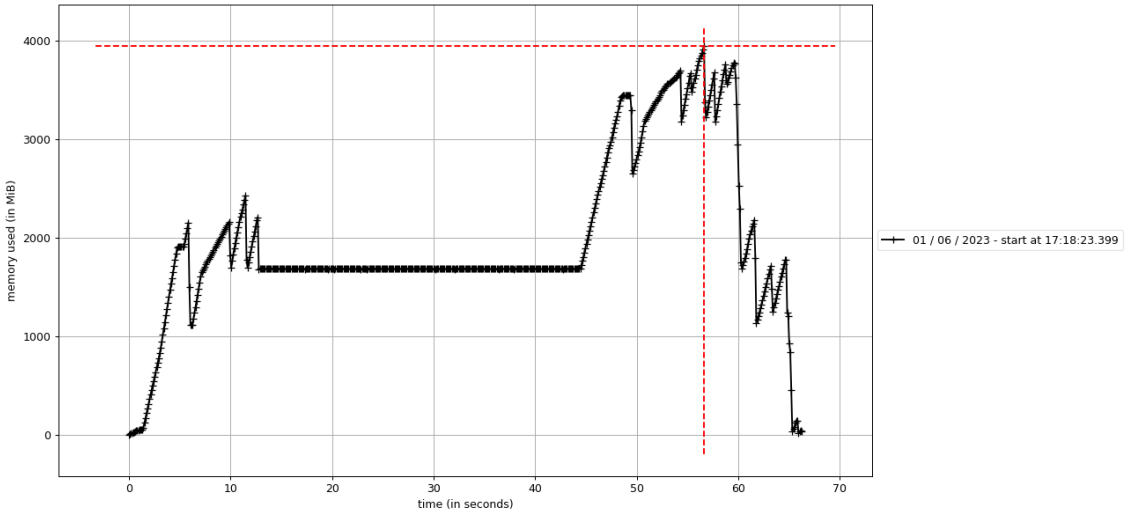
Ennek ellenére az algoritmus gyakorlatilag használhatatlan, mivel a két egymásbaágyazott *„for”* ciklus miatt nagyon lassú lesz. Hasonlítsuk össze két Python program futásidejét. Az egyik program használja ezt a logikát, míg a másik az 5.4.1-es fejezetben leírtat. Ezt mutatja be a következő, 9.1. ábra.

1	148,27	61,88
2	173,94	61,72
3	170,41	60,95
4	167,05	57,28
5	171,33	60,03
6	155,6	58,71
7	168,92	60,71
8	152,46	57,51
9	171,64	57,36
10	170,5	58,56
11	165,012	59,471

9.1. ábra: Futásidők összehasonlítása

Az első oszlopban a *„for”* ciklusokat használó program, a másodikban a *„get_BEM_matrices()”* függvényt használó program futásidejei láthatóak másodpercben. Mindkét programot tíz alkalommal futtattam le, és megmértem a futásidőket. A tizenegyedik, sárga hátterű sor az átlag időket tartalmazza. Ez alapján a *„for”* ciklusokat használó program átlagosan 165,012 másodperc alatt futott le, míg az 5.4.1-es fejezet logikáját használó program csupán 59,471 másodperc alatt. Ez a futásidő több, mint 2,77-szeres javulását jelenti.

Azonban ha megvizsgáljuk a *„get_BEM_matrices()”* függvényt (ld.: 5.4.1-es fejezet), akkor észrevehetjük, hogy immár nem csak a $\bar{\bar{G}}$ és $\bar{\bar{H}}$ mátrixok foglalnak jelentős mennyiségű memóriát. A Python programozási nyelv *„Memory Profiler”* nevű csomagja segítségével megjeleníthető memóriafelhasználást mutatja meg a 9.2. ábra.



9.2. ábra: A program memóriafelhasználása

Látható, hogy ugyanazzal a geometriával, amivel a *„for”* ciklusos megoldás 66,125 megabájt memóriát használt, ez a program majdnem 4000 megabájtot. Ez a szám a háromszöget számának növelésével négyzetesen növekszik.

Összefoglalva az egyik esetben túl hosszú volt a futásidő, míg a másik esetben túl sok volt a felhasznált memória. Az ideális megoldás valahol a kettő között lehet. Egy jó kompromisszum lehet, ha nem az egész mátrixot számítjuk ki egyszerre, és nem is csak egy elemét. Ha egyszerre egy sorát számítjuk ki a mátrixnak, tehát egy háromszög középpontja és minden háromszögön vett kvadratúrapontok között számolunk, akkor lassabb lesz az algoritmus, mint az 5.4.1-es fejezetben bemutatott, viszont jóval kevesebb memóriát fog felhasználni.

Irodalomjegyzék

[1] Dr. Rucz Péter: *Fundamental solutions and Green's functions*, Motivation https://last.hit.bme.hu/download/theoretical_acoustics/lecture_notes/01_fundamental.pdf (2023. 06. 04.)

[2] Dr. Rucz Péter: *Fundamental solutions and Green's functions*, Physical meaning https://last.hit.bme.hu/download/theoretical_acoustics/lecture_notes/01_fundamental.pdf (2023. 06. 04.)

[3] Python Software Foundation: *General Python FAQ*, https://docs.python.org/3/faq/general.html#what-is-python (2023. 05. 18.)

[4] ELTE IK, Programozási Nyelvek és Fordítóprogramok Tanszék: *A Python programozási nyelv*, http://nyelvek.inf.elte.hu/leirasok/Python/index.php?chapter=14 (2023. 05. 18.)

[5] The pip developers: *pip project*, https://pypi.org/project/pip/ (2023. 05. 18.)

[6] NumPy Developers: *User Guide*, https://numpy.org/doc/stable/user/whatisnumpy.html (2023. 05. 18.)

[7] NumPy Developers: *NumPy: the absolute basics for beginners*, https://numpy.org/doc/stable/user/absolute_beginners.html (2023. 05. 19.)

[8] The Matplotlib development team: *Matplotlib: Visualization with Python*, https://matplotlib.org/ (2023. 05. 19.)

[9] Python Code Quality Authority: *pylint project*, https://pypi.org/project/pylint/ (2023. 05. 20.)