POLITECHNIKA ŁÓDZKA
WYDZIAŁ BUDOWNICTWA, ARCHITEKTURY
I INŻYNIERII ŚRODOWISKA

KIERUNEK: BUDOWNICTWO

Damian Kozanecki
238125

MODELOWANIE DYNAMICZNEGO ZACHOWANIA PŁYT TYPU VIG Z WYKORZYSTANIEM SIECI NEURONOWYCH
MODELLING OF THE DYNAMIC BEHAVIOR OF VIG UNITS USING NEURAL NETWORKS

Praca dyplomowa II stopnia
napisana w Katedrze Mechaniki Konstrukcji
Promotor pracy
dr hab. inż. Artur Wirowski

ŁÓDŹ
2022

Mojemu Promotorowi, Panu dr. hab. inż. Arturowi Wirowskiemu, pragnę złożyć najserdeczniejsze wyrazy wdzięczności za nieocenioną pomoc przy realizacji niniejszej pracy. Za życzliwość, wyrozumiałość, wielkie wsparcie i ogrom cierpliwości, a także za merytoryczne rady i wskazówki, które z pewnością zaowocują w mojej przyszłej karierze.

Damian Kozanecki

Podziękowania

Praca została w części ufundowana przez Ministra Edukacji i Nauki RP na podstawie umowy nr SKN/SP/496315/2021.
Praca została wykonana z wykorzystaniem Infrastruktury PLGrid.

SPIS TREŚCI

1. Wprowadzenie

1.1. Znaczenie pracy

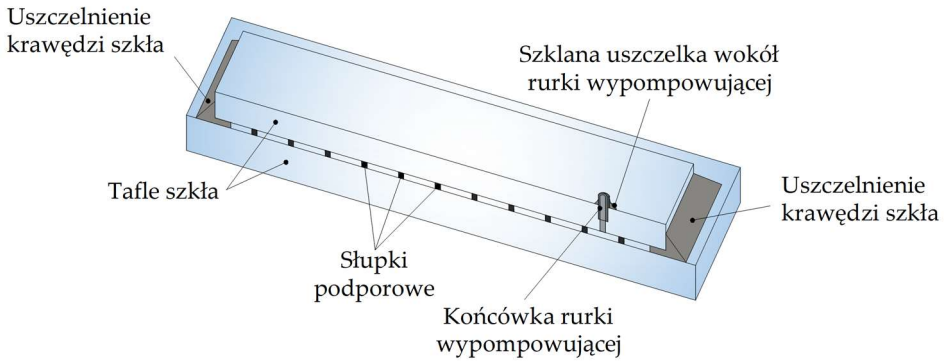
Wraz z niezwykle dynamicznym rozwojem przemysłowym na przełomie ostatnich lat, coraz wyraźniej dostrzec można zmieniający się na całym świecie klimat. Jednym z najistotniejszych czynników wywołujących to zjawisko jest stale rosnący poziom zużycia energii. Systematycznie na przestrzeni lat, dopuszczalna ilość energii zużywana na potrzeby ogrzania obiektów, w których przebywają ludzie, jest ograniczana. Wprowadzane są coraz bardziej rygorystyczne warunki dotyczące izolacyjności przegród zewnętrznych tych budynków. Jest to przyczyną stale rosnącego zapotrzebowania na bardziej energooszczędne rozwiązania konstrukcyjne.

Okna, będące nieodzowną częścią większości projektowanych budynków, są również jednym z najsłabszych elementów, jeśli chodzi o izolacyjność całej przegrody zewnętrznej. Najpopularniejszym rozwiązaniem są obecnie okna wieloszybowe wypełnione argonem lub innym gazem. Ciągły rozwój tej dziedziny doprowadził jednak inżynierów do zdecydowanie bardziej efektywnego pod względem izolacyjności rozwiązania, jakim jest szklenie próżniowe. VIG (ang. Vacuum Insulated Glass) to jedna z najbardziej zaawansowanych technologii, którą coraz częściej stosuje się w budownictwie.

Okna typu VIG nie są zupełnie nową technologią, ponieważ po raz pierwszy zostały opisane przez Zollera w literaturze patentowej w 1913 roku [1]. Od tego czasu technologia VIG była przedmiotem wielu badań i analiz. Mimo to, dopiero w 1989 roku pierwszy tego typu element został opracowany przez Collinsa, a następnie zaprezentowany na Kongresie ISES. Proces produkcji pierwszej płyty próżniowej okazał się bardzo czasochłonny i wymagający [2].

Proces produkcji płyt VIG i wprowadzania ich na rynek miał miejsce pod koniec lat 90-tych. Produkcja tego typu okien różni się wyraźnie od produkcji tradycyjnego okna kompozytowego [3]. Rysunek 1.1 przedstawia schemat budowy obecnie stosowanych paneli VIG.

Szklenie próżniowe VIG składa się z dwóch tafli szkła (obecnie stosuje się szkło hartowane o wysokiej wytrzymałości [4]) hermetycznie uszczelnionych na krawędziach za pomocą uszczelki. Ciśnienie atmosferyczne, panujące na zewnątrz szyby próżniowej powoduje ryzyko kontaktu tafli szkła ze sobą, co mogłoby również skutkować zniszczeniem tych elementów. Pomiędzy taflami umieszczono zatem system metalowych podpór – pilastrów [5], aby temu przeciwdziałać.



Rysunek 1.1. Schemat budowy okna VIG

Obecnie szyby próżniowe stosuje się najpowszechniej w sprzęcie AGD i konstrukcjach budowlanych. Zastosowanie tego typu szkła jako komponentu sprzętu AGD wynika z jego doskonałych właściwości fizycznych i estetycznych. W 2018 roku Siemens, Anthony i Haier zaprezentowali okno perspektywiczne typu VIG w lodówce. W efekcie zmiany oszklenia tradycyjnego na oszklenie próżniowe, zmniejszono grubość szyby i energochłonność tego urządzenia [6].

Okna VIG charakteryzuje niewielka grubość. Dzięki temu są one idealnym rozwiązaniem w przypadku modernizacji zabytkowego budynku. W tego typu obiektach wygląd okien musi wpisywać się w estetykę całego budynku. Wiele wiekowych budynków posiada nadal oryginalne stolarki okienne z pojedynczą szybą. Są one cienkie, ale mają znacząco wysoki współczynnik przenikania ciepła. Celem modernizacji tych obiektów jest zwiększenie efektywności energetycznej, poprzez jednoczesne zachowanie nienaruszonego wyglądu elewacji. Płyty VIG pozwalają na spełnienie tych kryteriów [6].

Pomimo, że wykorzystanie okien próżniowych w nowoprojektowanych budynkach nie jest jeszcze aż tak popularne jak w budynkach modernizowanych, mają one ogromny potencjał, aby podbić również ten sektor budowlany z uwagi na ich doskonałe parametry izolacyjne [6].

Jedną z głównych metod wykorzystanych w tej pracy jest metoda elementów skończonych (MES). Obecnie jest ona szeroko stosowana do analizy wielu problemów badawczych. Teotia i Soni wykorzystali MES do numerycznego odwzorowania złożonego zachowania zniszczenia warstwowego kompozytu polimerowego (szkło laminowane) [7]. Autorzy pracy [8] porównali wyniki doświadczalne i numerycznie zasymulowane zachowanie zginania trzech różnych płyt ze szkła laminowanego. Bedon w swojej pracy przedstawił dynamiczną charakterystykę istniejącej szklanej wiszącej kładki dla pieszych, a następnie wykonał model numeryczny MES w celu dodatkowej oceny i zbadania wydajności konstrukcji, w tym zbadania jej wrażliwości na oddziaływania zewnętrzne [9]. W przypadku płyt VIG, MES wykorzystano m.in. do badania wpływu różnych parametrów konstrukcyjnych pilastrów (przewodnictwo cieplne, geometria i rozmieszczenie) na efektywność cieplną tego rodzaju szyb [10].

1.2. Motywacja

Świetna izolacyjność szkła próżniowego została potwierdzona w wielu badaniach. W oparciu o badania numeryczne i laboratoryjne, Cho i Kim w swojej pracy zbadali współczynnik przenikania ciepła omawianych elementów [11], natomiast Ashmore, Cabrera i Kocer w swoich badaniach analizowali ich izolacyjność akustyczną [12]. Pomimo rosnącej popularności i wielu badań dotyczących płyt typu VIG, niewiele z istniejących prac dotyczy ich własności mechanicznych. Zhu rozważył wpływ pilastrów i uszczelnienia na termiczne i mechaniczne właściwości omawianych okien [10]. W badaniach tych wykorzystano zarówno metody numeryczne (MES) jak i metody eksperymentalne. Ponadto, dynamiczne zachowanie płyt typu VIG po raz pierwszy przeanalizowali Kowalczyk, Kozanecki, Krasoń i Rabenda w [13].

Biorąc pod uwagę złożoność analizowanego zagadnienia, wynikającą z wielu elementów składowych szyb próżniowych, można stwierdzić, że analiza dynamiczna może dostarczyć dużo więcej informacji na temat parametrów wytrzymałościowych wspomnianych elementów składowych okien w porównaniu z analizą statyczną. Jest to szczególnie istotne w przypadku szyb znajdujących się w istniejących konstrukcjach. Są one stale narażone na działanie czynników zewnętrznych takich jak promieniowanie UV, czy podmuchy wiatru. Właściwości mechaniczne każdego materiału z czasem mogą ulegać pogorszeniu. Ich weryfikacja w przypadku omawianych typów okien jest nie tylko istotna, ale również wymagająca. Ich demontaż i rozłożenie na części składowe sprawiłyby, że nie nadawałyby się one do dalszego użytku. Odpowiedzią na ten problem są nieniszczące badania dynamiczne.

1.3. Cel i zakres pracy

Niniejsza praca została zrealizowana w ramach projektu pt. „*Wykorzystanie sieci neuronowych w nieniszczących badaniach dynamicznych własności płyt kompozytowych*", realizowanego w ramach programu pod nazwą „*Studenckie koła naukowe tworzą innowacje*". Celem tego projektu było wykorzystanie sieci neuronowych do analizy dynamicznych własności płyt typu VIG, określenie zmian parametrów materiałowych pilastrów, znajdujących się pomiędzy taflami szkła, wraz z upływem czasu oraz optymalizacja ich rozmieszczenia. Zbiory danych, niezbędne do wykorzystania metod sztucznej inteligencji, pozyskano w oparciu o analizę numeryczną w oprogramowaniu ABAQUS CAE. Wyniki otrzymane z przeprowadzonych badań laboratoryjnych posłużyły do określenia efektywności modeli opartych o sieci neuronowe.

W poniższej pracy podjęto próbę przeanalizowania dynamicznego zachowania płyt typu VIG. Ponadto, w tej pracy do określenia parametrów wytrzymałościowych pilastrów znajdujących się pomiędzy taflami szkła wykorzystano metody sztucznej inteligencji.

Przyjęto następujące cele niniejszej pracy:

- znalezienie modelu numerycznego, który w sposób najbardziej efektywny i dokładny pozwoli odwzorować numerycznie rzeczywistą płytę typu VIG;
- przeprowadzenie analizy wpływu modułu sprężystości podłużnej, grubości tafli szkła oraz grubości warstwy próżni na wartości częstotliwości drgań własnych;
- przeprowadzenie analizy zależności kolejnych częstotliwości drgań własnych od siebie;
- określenie odpowiednich zbiorów danych, niezbędnych do przeprowadzenia analiz z wykorzystaniem metod sztucznej inteligencji;
- opis i dobór metod sztucznej inteligencji adekwatnych do analizowanego zagadnienia;
- określenie własności mechanicznych pilastrów znajdujących się w oknach typu VIG przy uwzględnieniu standardowych warunków podparcia okien występujących w rzeczywistych konstrukcjach;
- znalezienie poprawnego sposobu numerycznego odwzorowania schematu podparcia badanych laboratoryjnie płyt VIG;
- określenie własności mechanicznych pilastrów znajdujących się w oknach typu VIG w oparciu o wyniki z przeprowadzonych badań laboratoryjnych przy uwzględnieniu odpowiednich warunków podparcia;
- omówienie wniosków z przeprowadzonych analiz oraz określenie sposobów dalszego rozwoju proponowanych metod.

Pracę podzielono na cztery części. W pierwszej z nich przedstawiono zastosowane metody analizy, w drugiej przeanalizowano dynamiczne zachowanie płyt typu VIG, a w trzeciej zaprezentowane zostały zastosowane metody sztucznej inteligencji. Czwarta część jest natomiast wykorzystaniem trzech pierwszych, a zarazem podsumowaniem, w której wykorzystano przeanalizowane metody w badaniach laboratoryjnych. Poniżej, wspomniane części zostały szerzej omówione.

A. Metody analizy

W pierwszej części pracy opisano metody numeryczne wykorzystane do numerycznego odwzorowania rozważanego zagadnienia. Użyte zostały dwa różne oprogramowania – RFEM, w którym stworzono model powłokowo-prętowy oraz ABAQUS CAE, w którym stworzono model trójwymiarowy. Obydwa modele numeryczne wykorzystano do wyznaczenia częstotliwości drgań własnych.

W omawianym rozdziale przeprowadzono obliczenia numeryczne dla jednej płyty z ustalonymi parametrami geometrycznymi i mechanicznymi. W oparciu o wykonane obliczenia, omówiono dokładnie zastosowane rozwiązania oraz przyjęte założenia prowadzące do możliwie najdokładniejszego odwzorowania numerycznego tego zagadnienia. Przeanalizowano sposób numerycznego odwzorowania uszczelki i pilastrów oraz maksymalny rozmiar oczka siatki elementów skończonych. Opisano w nim również wady i zalety obydwu modeli oraz określono model przyjęty do dalszej analizy.

B. Analiza dynamicznego zachowania płyt typu VIG

W drugiej części pracy przeanalizowano dynamiczne zachowanie szyb próżniowych w oparciu o wartości częstotliwości drgań własnych dla różnych parametrów geometrycznych oraz własności mechanicznych płyt typu VIG z wykorzystaniem przyjętego uprzednio modelu numerycznego. Określono również znaczenie tych cech pod względem wpływu na wyznaczone wartości własne. Analizy przeprowadzono w oparciu o wyniki uzyskane przy uwzględnieniu standardowych warunków podparcia okien występujących w rzeczywistych konstrukcjach, na podstawie 360 różnych modeli numerycznych, których dokładne parametry opisano w rozdziale 4. Na koniec opisano otrzymane wyniki.

C. Metody sztucznej inteligencji

W trzeciej części pracy opisano wykorzystane metody sztucznej inteligencji oraz otrzymane na ich podstawie wyniki. W pierwszym kroku skategoryzowano wykorzystane metody. Następnie określono, kluczowe dla metod sztucznej inteligencji, przyjęte do analizy zbiory danych. W kolejnym kroku opisano metody Extreme Gradient Boosting (XGB) oraz Deep Neural Network (DNN), przy użyciu których stworzono modele predykcyjne. W ostatnim kroku przy użyciu stworzonych modeli aproksymowano poszukiwane wartości parametrów wytrzymałościowych pilastrów przy uwzględnieniu standardowych warunków podparcia okien występujących w rzeczywistych konstrukcjach.

D. Nieniszczące badania laboratoryjne z wykorzystaniem metod sztucznej inteligencji

W ostatniej części pracy wykorzystano metody użyte w pozostałych częściach pracy w celu zweryfikowania ich efektywności w nieniszczących badaniach laboratoryjnych. W ramach odrębnej pracy, której streszczenie opublikowano w [14], przeprowadzono badania laboratoryjne i wyznaczono częstotliwości drgań własnych testowanych szyb próżniowych. W niniejszej pracy opisano stworzony model numeryczny odwzorowujący sytuację eksperymentalną, określono zbiory danych oraz parametry metod sztucznej inteligencji. Finalnie, w oparciu o dane uzyskane w badaniach laboratoryjnych, określono parametry wytrzymałościowe pilastrów znajdujących się w badanych elementach.

2. Metody analizy – metoda elementów skończonych

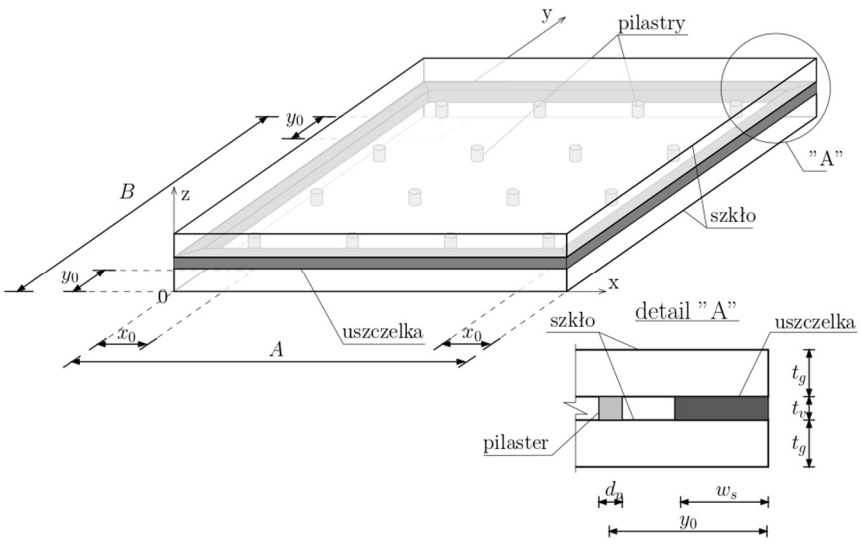
2.1. Opis analizowanego zagadnienia

2.1.1. Opis zadania

W celu odpowiedniego zamodelowania dynamicznego zachowania płyt typu VIG wykorzystano dwa różne modele numeryczne, a następnie zestawiono i porównano otrzymane wyniki. W celu wyznaczenia pierwszych trzydziestu częstotliwości drgań własnych płyty typu VIG, stworzono następujące modele numeryczne: model powłokowo-prętowy (przy użyciu oprogramowania RFEM) oraz model trójwymiarowy (przy użyciu oprogramowania ABAQUS CAE). Wszystkie analizy przeprowadzono dla płyty o ustalonych parametrach geometrycznych oraz mechanicznych (Tablica 2.1).

2.1.2. Geometria analizowanego elementu

Rzeczywistą płytę typu VIG sprowadzono do geometrycznie uproszczonego schematu na potrzeby tworzonych w dalszej części niniejszego rozdziału modeli numerycznych. Odwzorowanie geometryczne wykorzystane w niniejszej pracy składa się z dwóch elementów prostopadłościennych, znajdujących się pomiędzy nimi ortogonalnie rozstawionych elementów cylindrycznych oraz płaskich elementów obwodowych, które numerycznie odwzorowują kolejno dwie tafle szkła, nośne słupki podporowe (pilastry) oraz uszczelkę znajdującą się przy krawędzi wspomnianych tafli (Rysunek 2.1).



Rysunek 2.1 Odwzorowanie geometryczne płyty VIG

Dla opisanych w tym rozdziale modeli numerycznych, przyjęto następujące założenia związane z geometrią płyty VIG oraz właściwościami materiałowymi jej elementów składowych (Tablica 2.1).

Tablica 2.1 Zestawienie parametrów geometrycznych i materiałowych, przyjętych do obliczeń w rozdziale 2

Oznaczenie	Wartość	Jednostka	Opis
A	*0.60*	*m*	wymiar płyty w kierunku X
B	*0.40*	*m*	wymiar płyty w kierunku Y
t_g	*4.0*	*mm*	grubość tafli szkła
t_v	*0.3*	*mm*	grubość warstwy próżni
w_s	*9.0*	*mm*	szerokość uszczelki
n_x	*6*	-	ilość słupków w kierunku X
n_y	*10*	-	ilość słupków w kierunku Y
d_p	*0.6*	*mm*	średnica pilarka
x_p	*62.5*	*mm*	współrzędna X pierwszego pilarka
y_p	*52.5*	*mm*	współrzędna Y pierwszego pilarka
ρ_g	*2500.0*	*kg/m³*	gęstość szkła
E_g	*72.0*	*GPa*	moduł sprężystości podłużnej szkła
ν_g	*0.22*	-	współczynnik Poissona szkła
ρ_p	*7850.0*	*kg/m³*	gęstość stali (materiał słupków)
ν_p	*0.31*	-	współczynnik Poissona stali (materiał słupków)
E_p	*210.0*	*GPa*	moduł sprężystości podłużnej stali (materiał słupków)
ρ_s	*7850.0*	*kg/m³*	gęstość stali (materiał uszczelki)
E_s	*210.0*	*GPa*	moduł sprężystości podłużnej stali (materiał uszczelki)
ν_s	*0.31*	-	współczynnik Poissona stali (materiał uszczelki)

Jako warunki brzegowe rozważanego problemu, przyjęto zamocowanie płyty na wszystkich krawędziach. Zabieg ten zastosowano celem uwzględnienia standardowych warunków podparcia okien występujących w rzeczywistych konstrukcjach. Szczegóły realizacji warunków podparcia różnią się w zależności od rodzaju stosowanego modelu numerycznego (powłokowo-prętowy bądź trójwymiarowy) i zostały opisane w kolejnych rozdziałach.

2.2. RFEM – model prętowo-powłokowy

2.2.1. Opis oprogramowania

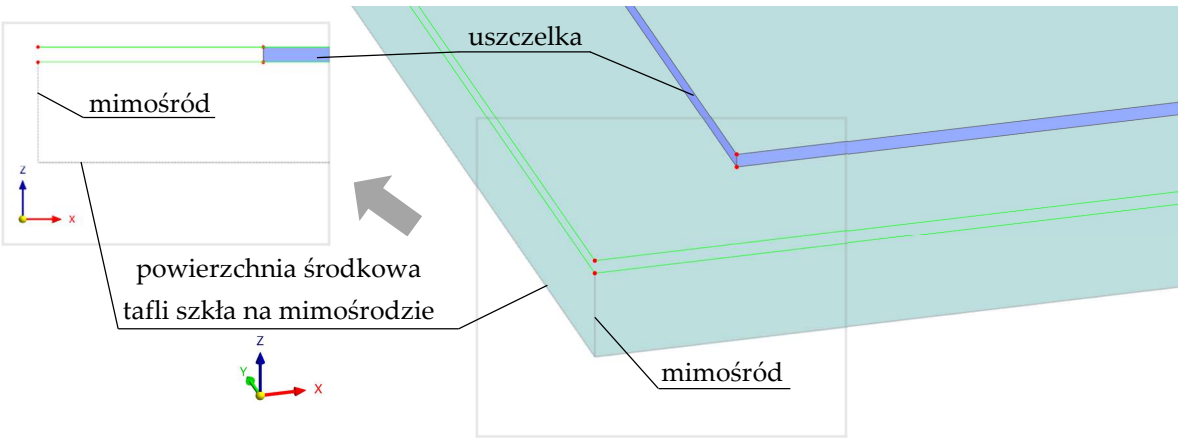
Pierwszą z metod wykorzystanych, w tej pracy, do stworzenia modelu numerycznego omawianej szyby typu VIG jest oprogramowanie RFEM 6.02 firmy Dlubal Software [15] oparte na metodzie elementów skończonych. Do rozwiązania poniżej omówionego zadania wykorzystaną wersję studencką programu.

2.2.2. Opis modelu

W modelu powłokowo-prętowym tafle szkła odwzorowano numerycznie jako dwuwymiarowe powłokowe elementy skończone. Odległość, która je dzieli jest równa grubości warstwy próżni, a co za tym idzie, również wysokości słupków podporowych. Biorąc pod uwagę wykorzystane do odwzorowania tafli szkła elementy skończone (elementy 2D), których powierzchnia stanowi odniesienie do płaszczyzny środkowej odwzorowywanych elementów, określono dla tych elementów mimośród o wartości połowy grubości tafli szkła. Mimośród ten wywiera określony wpływ na siły wewnętrzne powierzchni w postaci dodatkowych momentów. Rysunek 2.8 ilustruje omówiony powyżej szczegół.

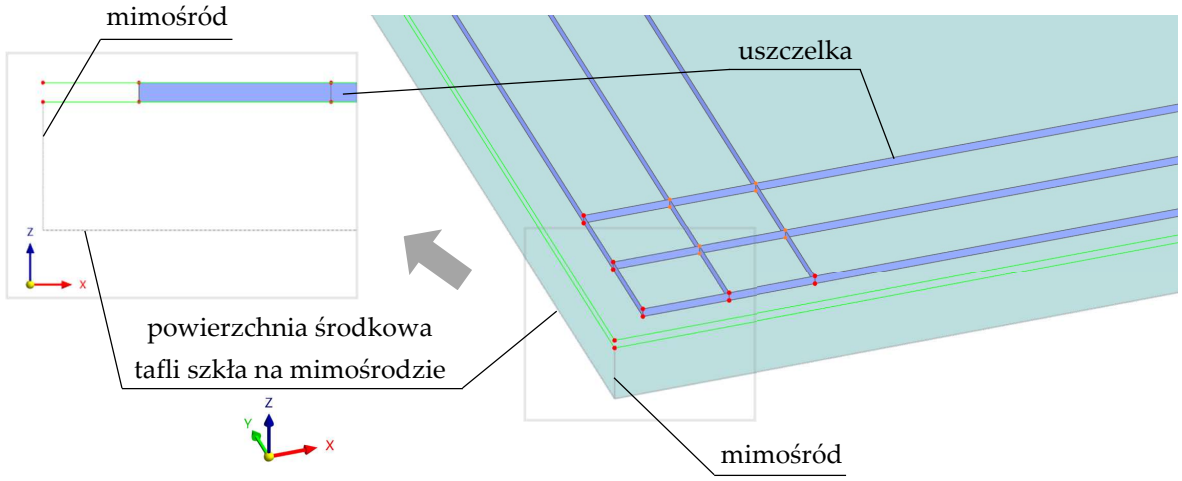
Uszczelkę odwzorowano w modelu numerycznym jako krzyżujące się ze sobą (tworzące siatkę – „układ ścian") dwuwymiarowe powłokowe elementy skończone. Z uwagi na przyjęcie modelu powłokowo-prętowego, numeryczne odwzorowanie elementu, którego długość (391 mm lub 591 mm) jest znacząco większa od wymiarów jego przekroju (9,0 mm x 0,3 mm), a zarazem łączącego swoimi powierzchniami bocznymi dwa inne elementy powłokowe, nie jest zagadnieniem trywialnym. Poniżej opisano analizę tego problemu oraz określono finalnie przyjęte rozwiązanie.

Określono trzy różne możliwe rozwiązania, a następnie porównano otrzymane wyniki. Pierwszym z nich było przyjęcie prostokątnych elementów powłokowych o wymiarach 391 mm x 0,3 mm i 591 mm x 0,3 mm oraz nadanie im cech wytrzymałościowych powłoki o grubości 9 mm. Elementy te ustawiono w osiach uszczelek w odniesieniu do rzutu całej płyty VIG. Rysunek 2.2 przedstawia widok z modelu obliczeniowego ukazujący zbliżenie naroża płyty VIG wraz z odwzorowaniem numerycznym uszczelek.



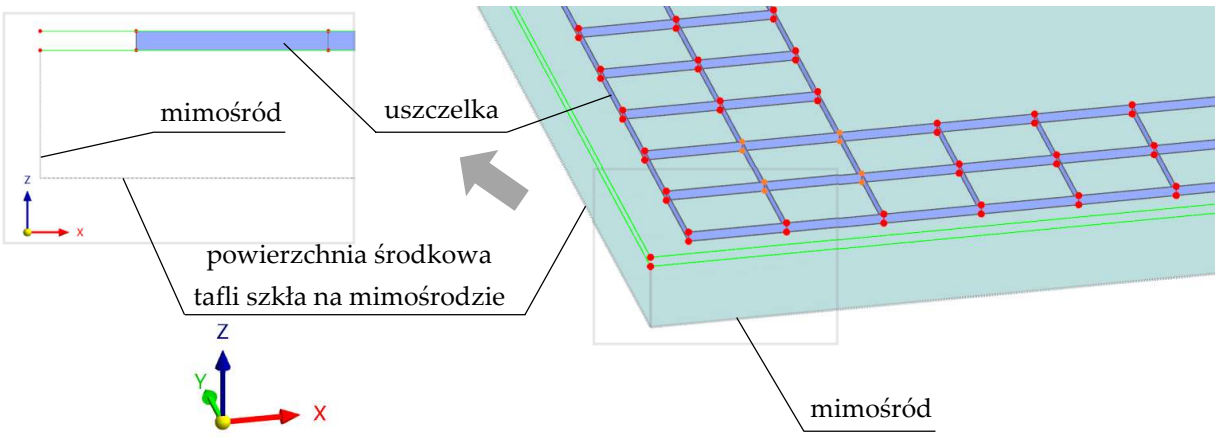
Rysunek 2.2 Zbliżenie naroża płyty VIG dla pierwszego sposobu numerycznego odwzorowania uszczelki

Drugim rozwiązaniem było podzielenie pojedynczych elementów na trzy osobne i nadanie im cech wytrzymałościowych elementu powłokowego o grubości 3 mm. Element środkowy znajduje się w miejscu elementu, który opisano wcześniej dla pierwszego sposobu odwzorowania tego elementu, a dwa dodatkowe odsunięto od niego o 3 mm, każdy w przeciwną stronę. Rysunek 2.3 przedstawia widok z modelu obliczeniowego ukazujący zbliżenie naroża płyty VIG wraz z odwzorowaniem numerycznym uszczelek.



Rysunek 2.3 Zbliżenie naroża płyty VIG dla drugiego sposobu numerycznego odwzorowania uszczelki

Trzeci sposób polega na dodaniu do elementów z drugiego sposobu, krótkich elementów powłokowych rozstawionych co 3 mm i prostopadłych do tamtych elementów. Nowododanym elementom również nadano cechy wytrzymałościowe elementów powłokowych o grubości 3 mm. Rysunek 2.4 przedstawia widok z modelu obliczeniowego ukazujący zbliżenie naroża płyty VIG wraz z odwzorowaniem numerycznym uszczelek.



Rysunek 2.4 Zbliżenie naroża płyty VIG dla trzeciego sposobu numerycznego odwzorowania uszczelki

Tablica 2.2 przedstawia zestawienie wyników otrzymanych przy użyciu tych sposobów.

Tablica 2.2 Częstotliwości własne otrzymane dla różnych sposobów numerycznego odwzorowania uszczelki

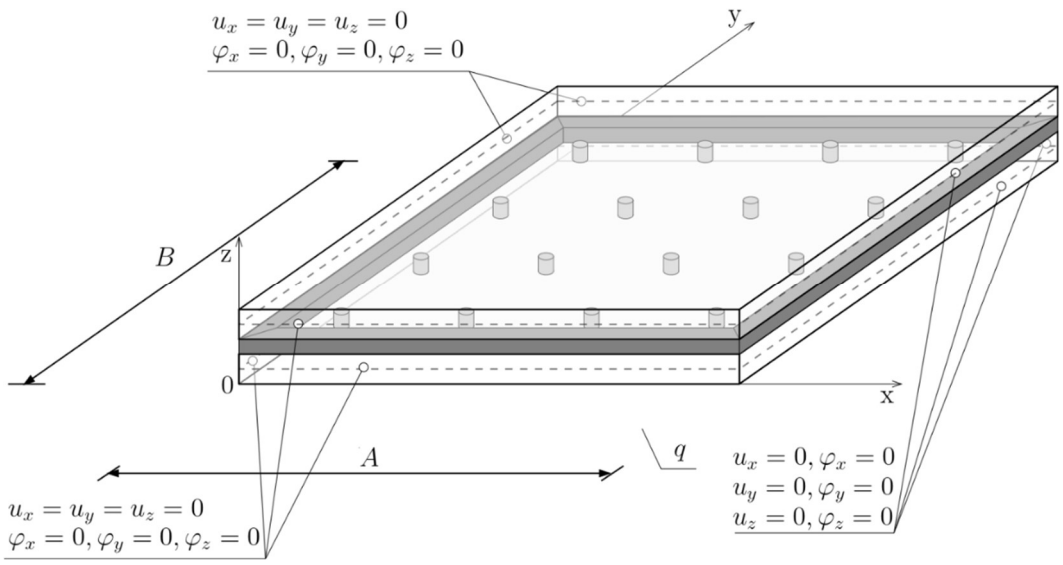
Nr częstotliwości drgań własnych	Częstotliwość dla danego sposobu [Hz]			Różnica pomiędzy modelami		
	I	II	III	I oraz II	I oraz III	II oraz III
1	196.42	200.05	200.11	1.8%	1.9%	0.0%
2	295.96	300.99	301.04	1.7%	1.7%	0.0%
3	458.19	467.24	467.37	2.0%	2.0%	0.0%
4	459.88	468.18	468.37	1.8%	1.8%	0.0%
5	547.75	558.90	559.06	2.0%	2.1%	0.0%
6	684.20	694.93	695.12	1.6%	1.6%	0.0%
7	699.43	712.54	712.83	1.9%	1.9%	0.0%
8	849.33	867.96	868.50	2.2%	2.3%	0.1%
9	913.17	929.31	929.71	1.8%	1.8%	0.0%
10	934.97	954.62	955.14	2.1%	2.2%	0.1%

Jako najlepsze numeryczne odwzorowanie uszczelki w modelu obliczeniowym przyjęto sposób drugi. Otrzymane za jego pomocą wyniki są zdecydowanie lepsze niż w przypadku sposobu pierwszego, a przy tym niemalże jednakowe jak w przypadku sposobu trzeciego. Ponadto, sposób drugi jest zdecydowanie mniej skomplikowany od sposobu trzeciego. Wybrany sposób został wykorzystany w dalszej analizie.

Słupki wsporcze odwzorowano w tym modelu jako jednowymiarowe prętowe elementy skończone typu belkowego. Z uwagi na kluczowe znaczenie tych elementów w odniesieniu do analizy dynamicznej całej płyty VIG, ważnym aspektem w procesie tworzenia modelu numerycznego było odpowiednie odwzorowanie połączenia elementu prętowego odwzorowującego słupek wsporczy oraz elementu płytowego odwzorowującego taflę szkła. W tym celu, sztywność obrotową tego połączenia w modelu numerycznym przyjęto jako skończoną. Obliczenia przeprowadzono kilkukrotnie dla różnych wartości tej sztywności. Wyniki uzyskane w ten sposób przedstawiono w punkcie 2.4 niniejszego rozdziału, celem porównania ich z wynikami otrzymanymi z modelu wykonanego w oprogramowaniu ABAQUS CAE.

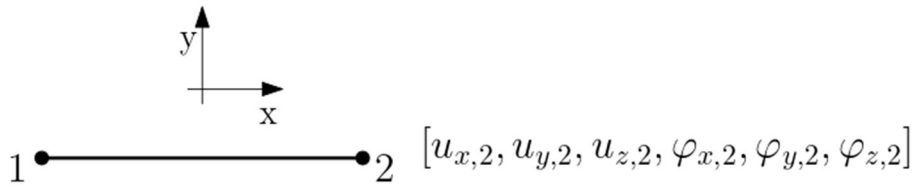
Materiały dla każdego z elementów składowych płyty VIG przyjęto jako liniowo-sprężyste, czego wynikiem było określenie wartości modułu Younga i współczynnika Poissona. Dodatkowo, ustalono również gęstości materiałów, które są niezbędne do przeprowadzenia analizy dynamicznej.

Celem odwzorowania zamocowania płyty na wszystkich jej krawędziach, sposób podparcia przyjęto jako odebranie wszystkich sześciu stopni swobody w węzłach znajdujących się na krawędziach powłokowych elementów skończonych odwzorowujących numerycznie tafle szkła. Zablokowane zostały zatem zarówno przemieszczenia wzdłuż osi x, y, z (u_x, u_y i u_z), jak i obroty względem tych osi (φ_x, φ_y i φ_z) (Rysunek 2.5).



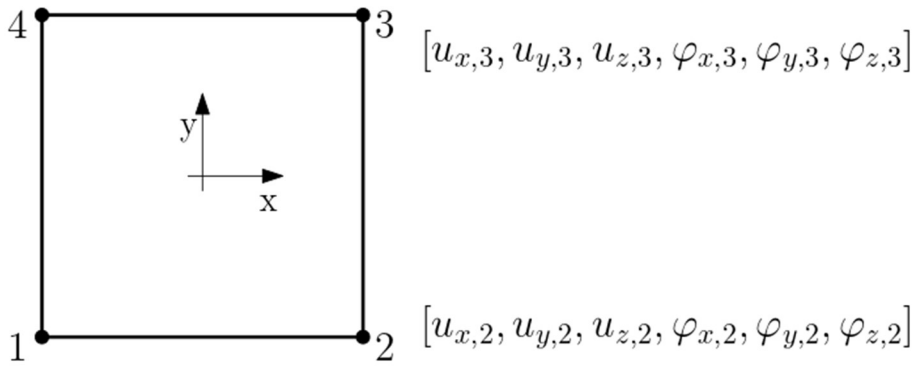
Rysunek 2.5 Przyjęte w modelu numerycznym warunki brzegowe (RFEM)

Przyjęte w programie RFEM jednowymiarowe elementy skończone są typowymi elementami skończonymi typu 1D, mającymi dwanaście stopni swobody – po sześć na każdym z końców elementu – są to przemieszczenia (u_x, u_y, u_z) oraz obroty (φ_x, φ_y, φ_z). W przypadku tych elementów, zakłada się, że przekrój po odkształceniu pozostaje płaski.



Rysunek 2.6 Przyjęty jednowymiarowy element skończony (RFEM) [16]

Drugim typem elementów skończonych wykorzystywanych w tej analizie są dwuwymiarowe elementy powłokowe. Automatyczny algorytm, generujący siatkę elementów skończonych, w miejscach w których nie jest możliwe zastosowanie elementów czworokątnych stosuje elementy trójkątne. Stopnie swobody w węzłach narożnych elementów czworokątnych oraz trójkątnych są jednakowe jak dla elementów jednowymiarowych. Dzięki temu zapewniona jest kompatybilność elementów jednowymiarowych oraz dwuwymiarowych w węzłach.

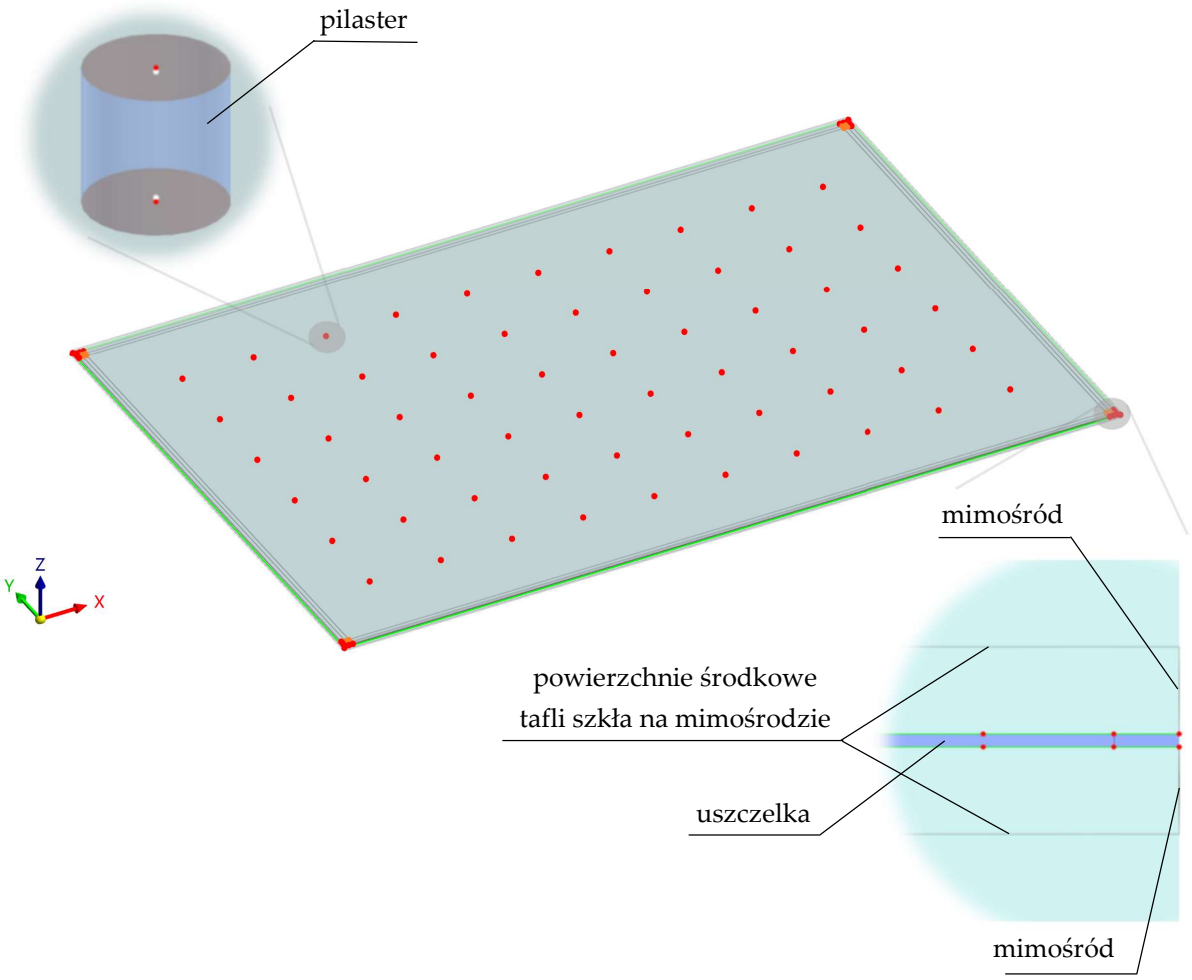


Rysunek 2.7 Przyjęty dwuwymiarowy element skończony (RFEM) [16]

W celu wyznaczenia częstotliwości drgań własnych należy rozwiązać następujące zagadnienie własne:

$$(-\omega^2 \boldsymbol{M}^{MN} + \boldsymbol{K}^{MN})\boldsymbol{\phi}^N = 0 \tag{2.1}$$

gdzie $\boldsymbol{M}^{MN}$ to macierz mas, $\boldsymbol{K}^{MN}$ to macierz sztywności, $\boldsymbol{\phi}^N$ to wektor własny (postać drgań), a M i N to stopnie swobody. Program RFEM pozwala wykorzystać jedną z trzech następujących metod wyznaczania poszukiwanych wartości własnych: metoda Lanczosa, metoda pierwiastka wielomianu charakterystycznego oraz metoda iteracji podprzestrzennej. W niniejszej pracy wybrano metodę pierwiastka wielomianu charakterystycznego wraz z zestawem 30 wartości własnych.
Rysunek 2.8 przedstawia widok modelu numerycznego opisanego w niniejszym punkcie rozdziału.



Rysunek 2.8 Widok modelu numerycznego stworzonego w programie RFEM

2.2.3. Siatkowanie

W celu dobrania odpowiedniego rozmiaru oczka siatki elementów skończonych modelu numerycznego, przeprowadzono obliczenia dla siatek o następujących maksymalnych wymiarach elementów skończonych: 0.05 m; 0.04 m; 0.03 m; 0.02 m; 0.01 m; 0.005 m; 0.002 m. Tablica 2.1 przedstawia pierwsze dziesięć wartości częstotliwości drgań własnych dla tych wartości. Ponadto, podano również czas obliczeń dla każdego z przypadków. Obliczenia przeprowadzono przy założeniu nieskończonej sztywności obrotowej węźle łączącym element odwzorowujący numerycznie słupek podporowy z elementem odwzorowującym numerycznie taflę szkła.

Tablica 2.3 Wyniki otrzymane dla różnych rozmiarów siatki elementów skończonych (RFEM)

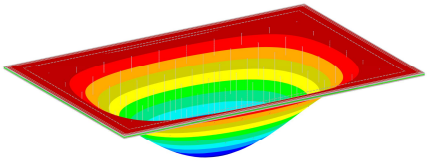
Nr częstotliwości drgań własnych	Częstotliwość drgań własnych [Hz] dla danego rozmiaru siatki elementów skończonych [m]						
	0.050	0.040	0.030	0.020	0.010	0.005	0.002
1	205.54	206.38	203.62	201.64	200.05	198.98	197.84

2	317.04	311.52	307.18	303.52	300.99	299.45	297.94
3	492.90	492.09	480.72	473.34	467.24	464.59	462.59
4	502.23	496.40	486.28	475.02	468.18	465.13	463.18
5	554.49	593.07	578.78	566.23	558.90	555.65	553.54
6	610.78	639.45	726.50	708.70	694.93	689.89	687.14
7	766.83	752.37	738.71	723.47	712.54	708.59	706.17
8	801.24	767.36	775.03	880.00	867.96	859.84	856.38
9	897.62	981.30	928.50	891.51	929.31	923.33	920.31
10	949.52	1035.68	980.20	947.38	954.62	946.56	943.05
11	1084.90	1067.17	1017.03	978.79	981.99	972.92	968.97
12	1105.57	1093.70	1048.31	1011.29	1095.80	1092.85	1089.17
13	1119.14	1120.23	1170.41	1126.54	1101.00	1170.96	1193.90
14	1266.05	1231.84	1217.34	1141.52	1199.84	1198.04	1207.04
15	1278.03	1231.95	1284.21	1243.20	1207.57	1201.10	1234.74
16	1334.35	1387.60	1298.20	1292.52	1308.69	1299.12	1294.97
17	1457.06	1466.09	1401.69	1339.13	1329.09	1312.49	1306.62
18	1535.95	1523.90	1455.16	1384.47	1400.47	1380.29	1373.83
19	1546.81	1541.16	1522.21	1460.07	1485.21	1465.20	1458.73
20	1554.92	1663.56	1570.72	1544.81	1521.22	1530.88	1524.89
21	1615.56	1684.89	1654.06	1587.75	1547.79	1565.55	1560.49
22	1678.43	1737.67	1696.88	1609.13	1577.80	1607.73	1601.21
23	1798.23	1782.12	1708.89	1622.72	1627.46	1618.79	1655.86
24	1833.01	1785.49	1809.02	1640.90	1649.14	1653.88	1679.02
25	1973.52	1867.37	1809.47	1688.77	1734.91	1706.89	1698.06
26	2090.20	1892.17	1818.10	1813.27	1829.23	1808.53	1801.76
27	2093.14	1956.88	1827.41	1831.46	1900.77	1890.38	1883.63
28	2103.25	2083.17	1949.23	1892.42	1909.28	1920.06	1911.22
29	2146.67	2176.86	2040.61	1902.78	1947.90	2024.21	2012.33
30	2158.33	2215.30	2101.02	1978.43	2037.06	2051.53	2060.30
Czas obliczeń [s]	13	17	31	45	191	1238	12845

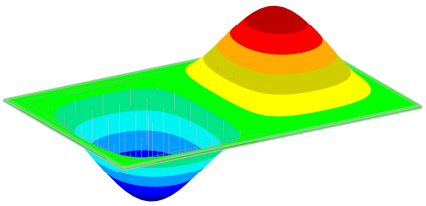
Na potrzeby dalszej analizy, jako maksymalny rozmiar oczka siatki elementów skończonych przyjęto 0.010 m. Jest to wystarczająco mały wymiar elementu, który pozwala uzyskać odpowiednio dokładny wynik przy możliwie najkrótszym czasie prowadzonych obliczeń.

2.2.4. Wyniki

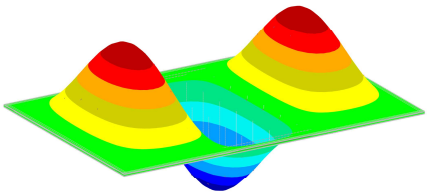
Dla modelu numerycznego uwzględniającego przyjęte w tym rozdziale założenia, wyznaczono trzydzieści pierwszych częstotliwości drgań własnych. Tablica 2.3 zawiera te wartości w kolumnie z wynikami dla siatki o rozmiarze 0.01 m. Rysunek 2.9 ilustruje powiązane z tymi wartościami postacie drgań własnych.



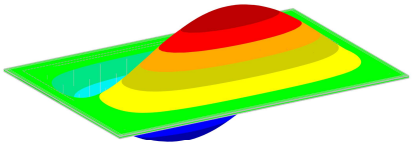
1



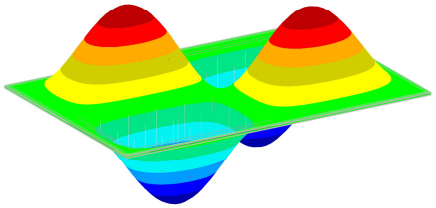
2



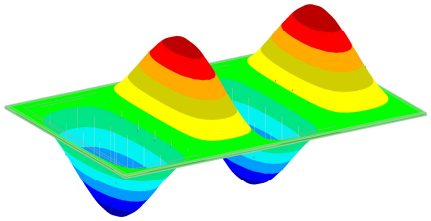
3



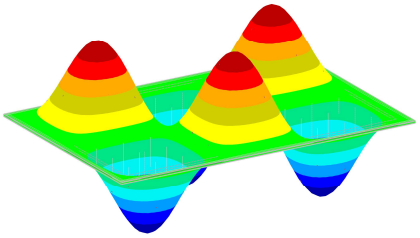
4



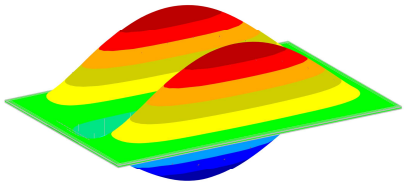
5



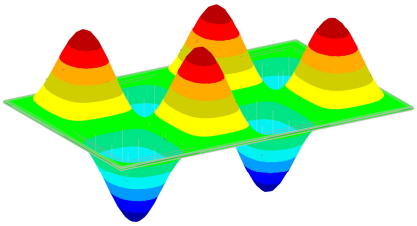
6



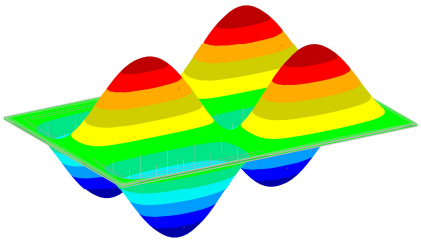
7



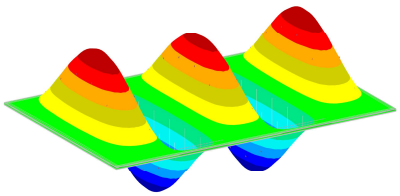
8



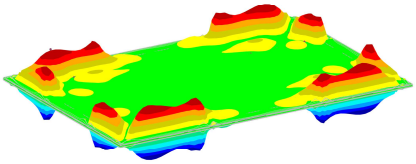
9



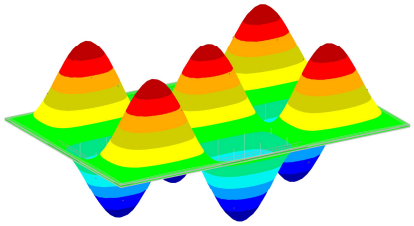
10



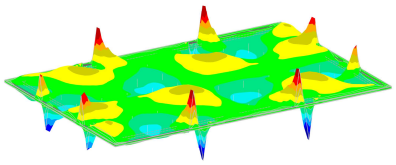
11



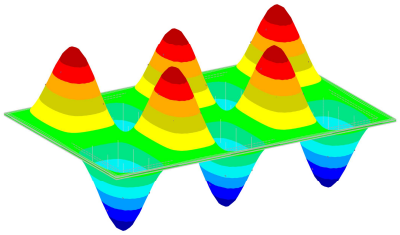
12



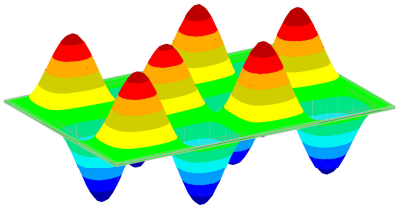
13



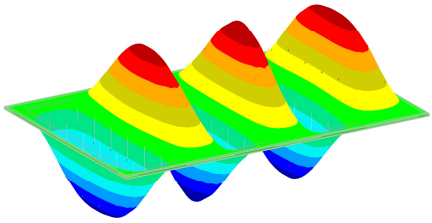
14



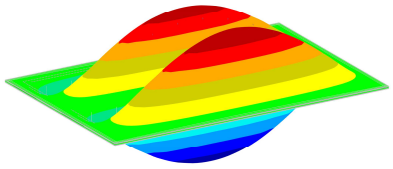
15



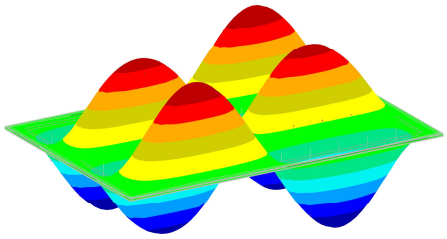
16



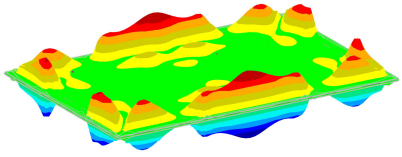
17



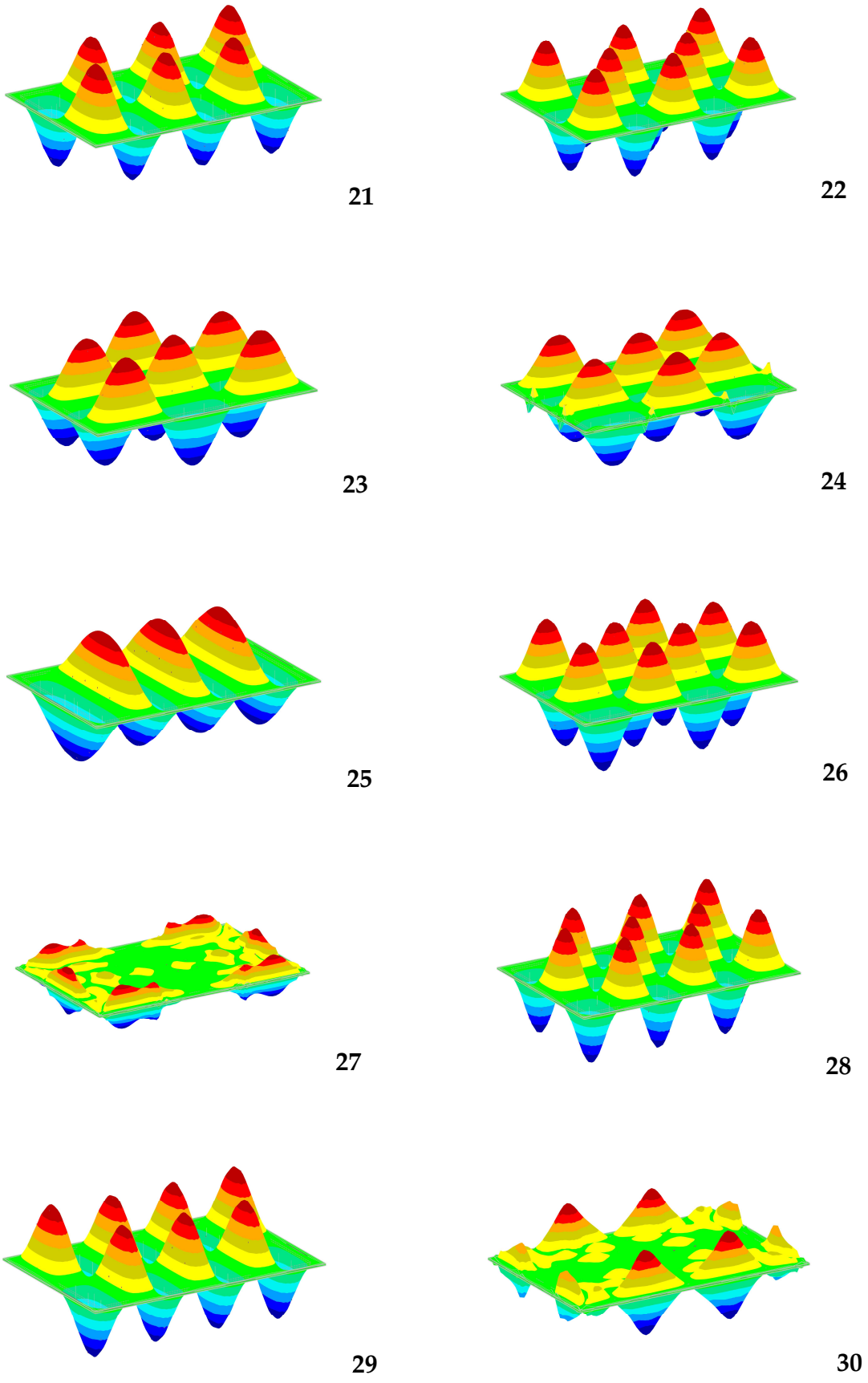
18



19



20



Rysunek 2.9 Postacie drgań własnych dla odpowiednio pierwszych trzydziestu częstotliwości drgań własnych – RFEM

2.3. ABAQUS – model przestrzenny

2.3.1. Opis oprogramowania

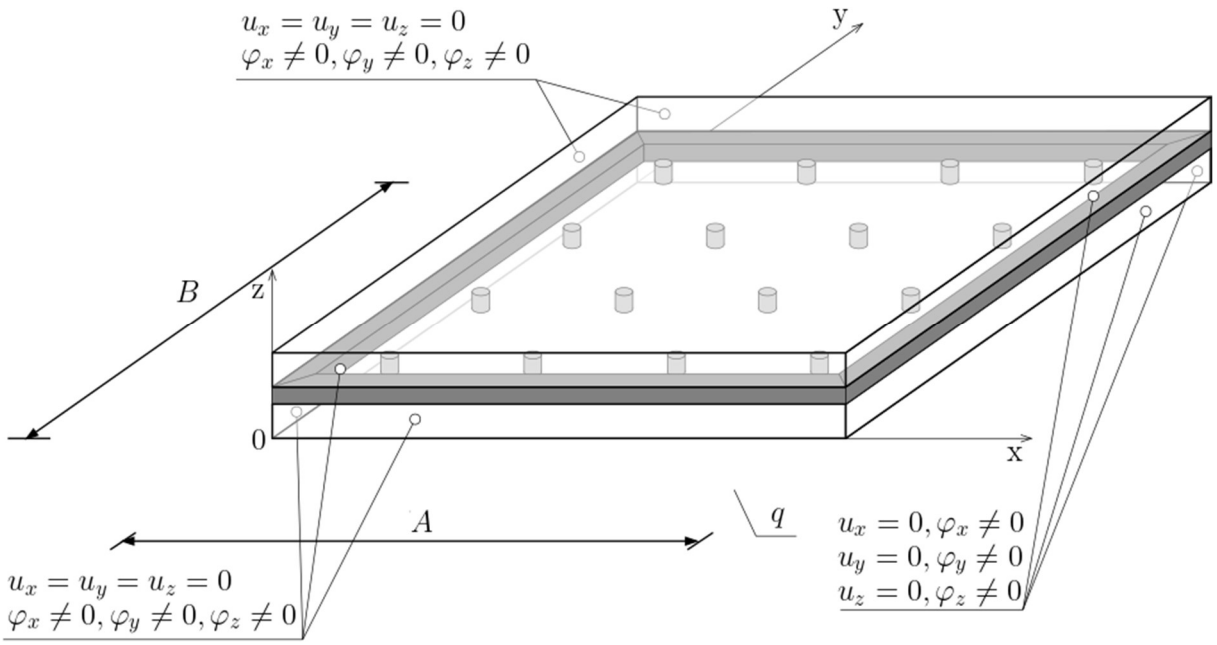
Drugą z metod, które wykorzystano w tej pracy, jest oprogramowanie ABAQUS CAE firmy Dassault Systèmes Simulia Corporation [17]. Obliczenia zostały wykonane przy użyciu Infrastruktury PLGrid, która zapewnia dostęp do klastrów zlokalizowanych w pięciu centrach wysokiej mocy obliczeniowej. Zadania obliczeniowe są zlecane na zewnątrz poprzez tzw. oprogramowanie pośredniczące, które centralnie zarządza zasobami wszystkich koncentratorów. W tej analizie wykorzystano węzeł WCSS (Wrocławskie Centrum Sieciowo-Superkomputerowe).

2.3.2. Opis modelu

W modelu numerycznym ABAQUS pilarki podporowe odwzorowano jako elementy trójwymiarowe, co zapewniło większą dokładność odwzorowania tych elementów [10], [18]. Przyjęty model pozwala na uwzględnienie koncentracji naprężeń w podporach oraz deformacji wynikających ze ścinania. W punkcie 2.3.3 przeanalizowano dobór rozmiaru siatki elementów skończonych. Wykazano, że jej właściwe przyjęcie w otoczeniu pilarka, lepiej odwzorowuje charakter jego odkształceń.

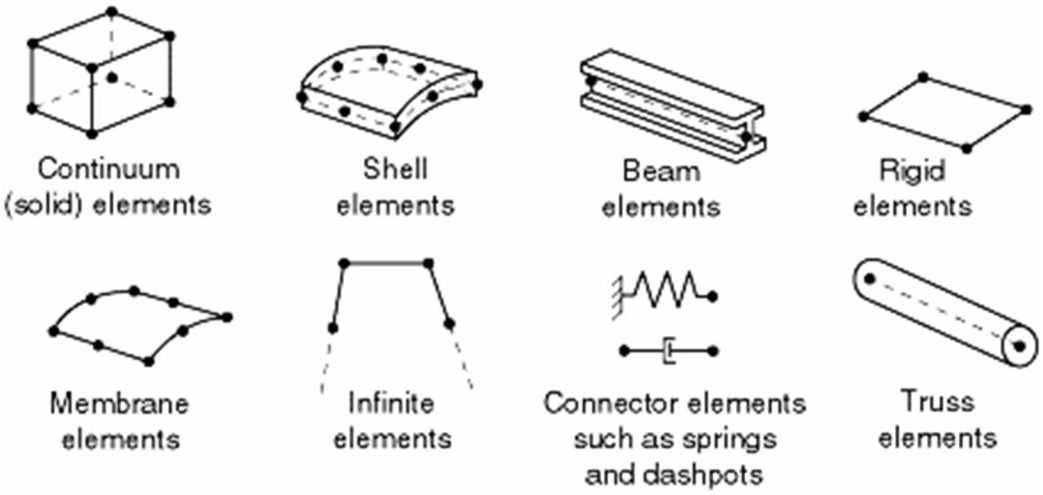
Ponadto, również wszystkie pozostałe elementy składowe szyby odwzorowano w modelu numerycznym przy użyciu trójwymiarowych elementów skończonych. Wszystkie elementy modelu (słupki, uszczelka i tafle szkła) zostały stworzone osobno. Materiały dla każdego z nich przyjęto jako liniowo-sprężyste, co zdeterminowało wprowadzenie modułu Younga i współczynnika Poissona. Dodatkowo, ustalono również gęstości materiałów, które są niezbędne w analizie dynamicznej. W celu połączenia ze sobą wszystkich stworzonych osobno części, zastosowano wiązanie, w którym powierzchnie główna i podrzędna są przypisane odpowiednio do pilastrów/uszczelek i szkła. Podczas łączenia dwóch powierzchni są one klasyfikowane, w wykorzystywanym oprogramowaniu, jako powierzchnia główna (ang. master) i podrzędna (ang. slave). Celem uzyskania możliwie najlepszej symulacji kontaktu, jako powierzchnię podrzędną przyjęto tę o mniejszym generalnym rozmiarze siatki elementów skończonych.

Schemat statyczny rozpatrywanej płyty typu VIG przyjęto jako płytę zamocowaną na krawędziach. Warunki brzegowe zostały przedstawione w modelu MES jako stałe przemieszczenia wzdłuż osi x, y, z (u_x, u_y i u_z) na bocznych powierzchniach elementów imitujących tafle szkła, ale stopnie swobody obrotu (φ_x, φ_y i φ_z) pozostawiono jako swobodne (Rysunek 2.10). Należy jednak nadmienić, że obrót na krawędziach w modelu prawie nie występuje, ponieważ podpory obejmują całą powierzchnię boczną, a nie tylko krawędź.



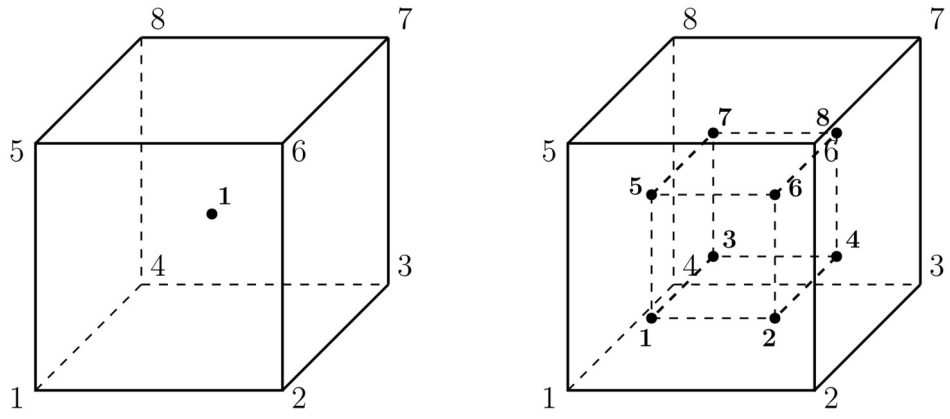
Rysunek 2.10 Przyjęte w modelu numerycznym warunki brzegowe (ABAQUS)

Istnieje pięć następujących parametrów charakteryzujących elementy w ABAQUS: rodzina, stopnie swobody (związane z rodziną elementów), liczba węzłów, sformułowanie oraz integracja. Dzięki ich określeniu, element otrzymuje nazwę identyfikującą każdy ze wspomnianych parametrów. Powszechnie używane rodziny elementów przedstawiono poniżej (Rysunek 2.11).



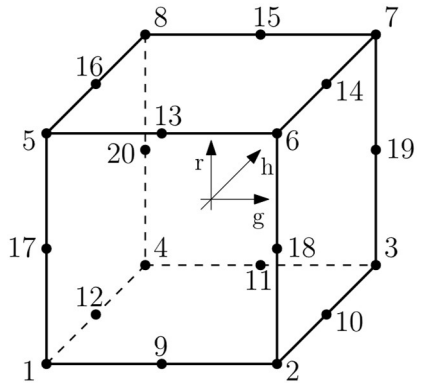
Rysunek 2.11 Zestawienie najczęściej używanych rodzin elementów w ABAQUS [19]

Stopniami swobody dla symulacji naprężeń/przemieszczeń są translacje i obroty. Biorąc pod uwagę liczbę węzłów i kolejność interpolacji, przemieszczenia są obliczane w węzłach elementu, a przemieszczenia w dowolnym innym punkcie są uzyskiwane przez interpolację z przemieszczenia węzłów. Przeważnie kolejność interpolacji zależy od liczby węzłów użytych w elementach. W przypadku 8-węzłowego elementu 3D rozróżnia się dwa typy: C3D8R i C3D8 przedstawione poniżej (Rysunek 2.12).



Rysunek 2.12 (a) element C3D8R o zredukowanej integracji (1 punkt integracji) zestawiony z elementem C3D8 (w pełni zintegrowany – 2 × 2 × 2 punkty integracji) [20]

W niniejszej pracy przyjęto elementy typu C3D8R (Rysunek 2.13) — element pierwszego rzędu o zredukowanej integracji z aktywowaną kontrolą klepsydry [20].



Rysunek 2.13 Oznaczenia węzłów w przyjętym typie elementu skończonego [19]

Funkcja interpolacji $\boldsymbol{u}$ dla elementu C3D8R jest określona za pomocą równania (2.2):

$$\boldsymbol{u} = N^I(g,h,r)\boldsymbol{u^I} \text{ suma na } \boldsymbol{I}, \tag{2.2}$$

gdzie N^I to izoparametryczne funkcje kształtu, a $\boldsymbol{I}$ to węzeł elementu.

Funkcje kształtu są takie same jak dla elementu typu C3D8 [19], [20] (2.3):

$$N^I(g,h,r) = \frac{1}{8}\Sigma^I + \frac{1}{4}g\Lambda_1^I + \frac{1}{4}h\Lambda_2^I + \frac{1}{4}r\Lambda_3^I + \frac{1}{2}hr\Gamma_1^I + \frac{1}{2}gr\Gamma_2^I + \frac{1}{2}gh\Gamma_3^I + \frac{1}{2}ghr\Gamma_4^I, \tag{2.3}$$

gdzie I oznacza węzeł elementu. Ostatnie cztery wektory Γ_α^I, α=1, 2, 3, 4 są bazowymi wektorami klepsydry. Macierz gradientu $\boldsymbol{B^I}$ definiuje się jako całkowanie po elemencie skończonym:

$$\boldsymbol{B}_i^I = \frac{1}{V_{el}} \int_{V_{el}} N_i^I\,(g,h,r) dV_{el,}\; N_i^I(g,h,r) = \frac{\partial N^I}{\partial x_i}, \tag{2.4}$$

gdzie V_{el} jest objętością elementu, a i = 1, 2, 3. W sformułowaniu odkształcenia centroidalnego macierz gradientu jest przedstawiona jako:

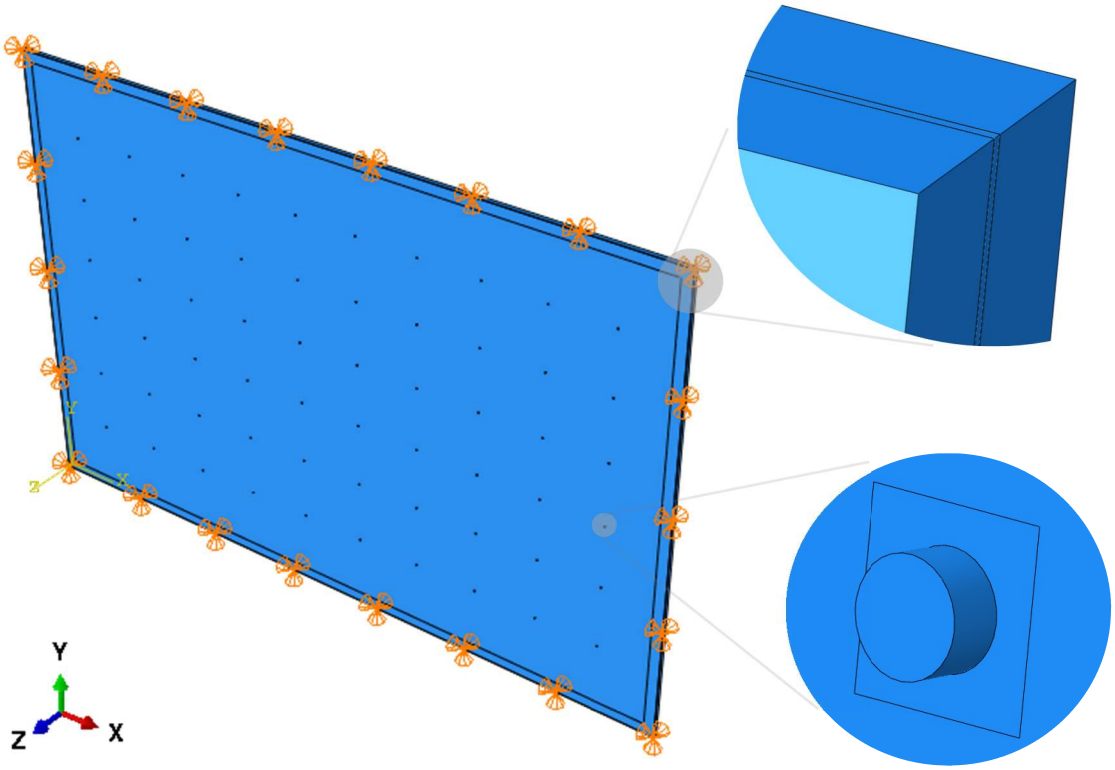
$$\boldsymbol{B}_i^I = N_i^I(0,0,0)\,. \tag{2.5}$$

Uwzględniając powyższe założenie, można zauważyć, że formułowanie odkształcenia centroidalnego upraszcza proces wyznaczania macierzy gradientu. W ABAQUS do kontroli trybów klepsydry w tych elementach, wykorzystywana jest metoda sztucznej sztywności i metoda sztucznego tłumienia pokazana w pracy [21]. Natomiast, w pracy [22] porównano ich skuteczność dla elementów C3D8R z kontrolą klepsydry w odniesieniu do elementów innych typów.

Ostatnim aspektem jest sformułowanie elementu, które odnosi się do teorii matematycznej używanej do określenia zachowania tego elementu. W ABAQUS wszystkie elementy typu naprężenie/odkształcenie są oparte na Lagranżowskim lub materiałowym opisie zachowania.

W celu przeprowadzenia analizy dynamicznej, utworzono krok zaburzenia liniowego. Odpowiedź na etapie analizy liniowej jest odpowiedzią na zaburzenia liniowe dotyczące stanu podstawowego (krok przed etapem zaburzeń liniowych). Etap wyznaczenia częstotliwości jest przeprowadzany poprzez wyznaczenie wartości własnych w celu obliczenia częstotliwości drgań własnych i odpowiednich kształtów postaci drgań danego modelu. Liniowość geometryczna została uwzględniona. Rozważono zagadnienie wartości własnej dla częstotliwości drgań własnych poprzez rozpatrzenie równania (2.1). ABAQUS daje możliwość wykorzystania jednej z trzech metod wyznaczania poszukiwanych wartości własnych: metoda Lanczosa (metoda domyślna), metoda AMS (Automatyczna podstrukturyzacja wielopoziomowa) oraz metoda iteracji podprzestrzennej. Wybrana została metoda Lanczosa wraz z zestawem 30 wartości własnych.

Rysunek 2.14 przedstawia widok modelu numerycznego opisanego w powyższym punkcie rozdziału.



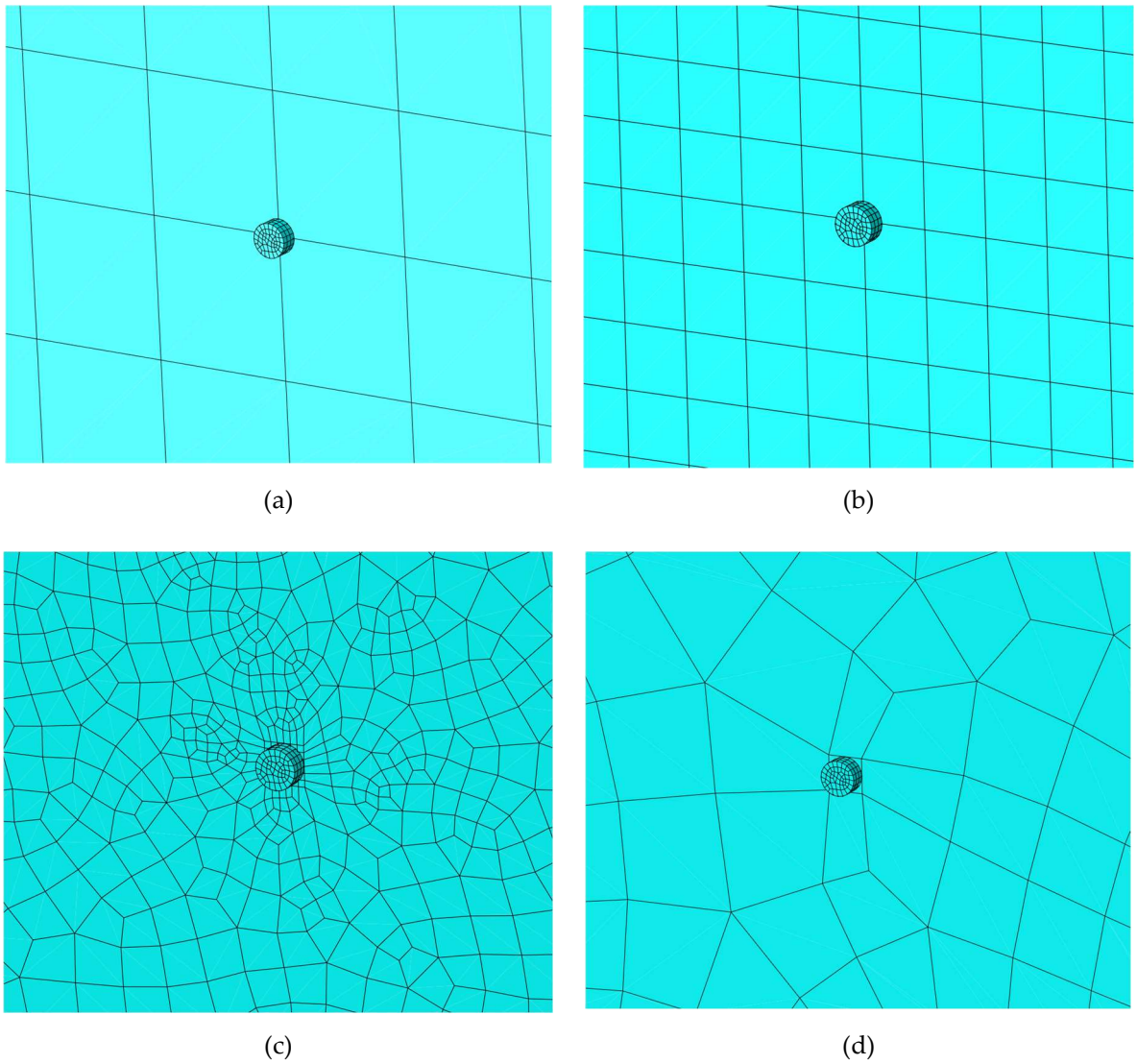
Rysunek 2.14 Widok modelu numerycznego stworzonego w programie ABAQUS

2.3.3. Siatkowanie

Analizowane elementy (płyty VIG) składają się z nośnych słupków podporowych, które w porównaniu do tafli szkła, są niemal mikroskopijne. Przy doborze odpowiedniego globalnego rozmiaru siatki elementów skończonych należy wziąć pod uwagę powierzchnie styku tafli szkła i słupków nośnych. Rozmiar siatkowania elementów odwzorowujących przeszklenie musi być większy niż rozmiar oczek słupków podporowych, aby zachować rozsądny rozmiar modelu numerycznego. Zgodnie z dokumentacją ABAQUS [19], węzły jednego elementu skończonego, znajdujące się wewnątrz innego elementu skończonego, są sztucznie połączone algorytmem z węzłami tego drugiego.

Poszukując najdokładniejszego numerycznego rozwiązania rozważanego problemu, stworzono modele testowe i założono różne gęstości siatkowania. Przewiduje się wystąpienie dwóch następujących typów drgań – w fazie oraz w antyfazie. Z uwagi na większą problematykę tych drugich, celem poprawnego wyznaczenia pierwszych częstotliwości drgań własnych, związanych z tego rodzaju drganiami, wykonano obliczenia dla wielkości oczek szyb: 2.5, 1.0, 1.0 (0.1 mm przy podporach) i 2.5 mm (0.5 mm w pobliżu słupków wsporczych) (Rysunek 2.15).

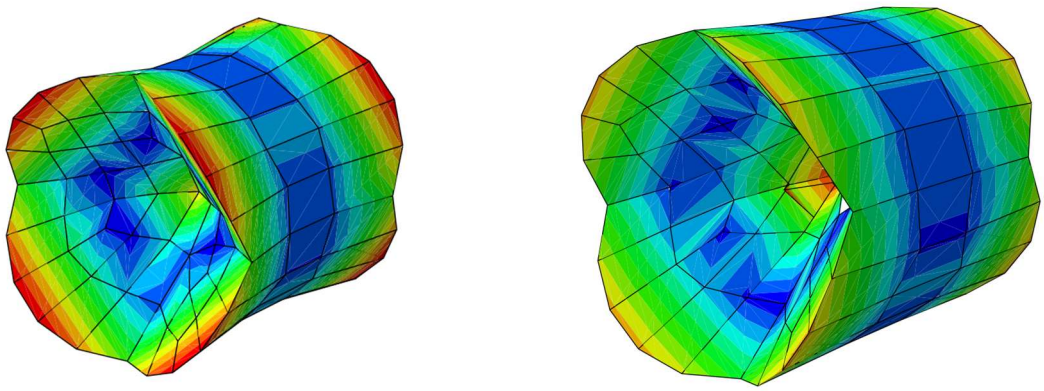
Kształty odkształceń słupków podporowych w pierwszych częstotliwościach własnych związanych z drganiami przesuniętymi w fazie, dla założonych rozmiarów oczek, przedstawiono w odpowiedniej kolejności na poniższym rysunku (Rysunek 2.16).



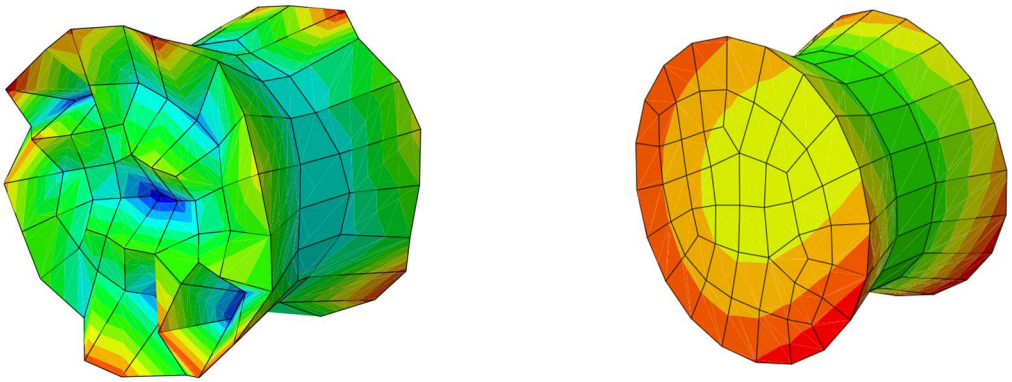
(a) (b)

(c) (d)

Rysunek 2.15 Rozmiar siatkowania tafli szkła w pobliżu słupka podporowego dla następujących rozmiarów oczek: a) 2.5 mm; (b) 1.0 mm; (c) 1.0 mm (0.1 mm w pobliżu słupków wsporczych); oraz (d) 2.5 mm (0.5 mm w pobliżu słupków wsporczych)



(a) (b)



(c) (d)

Rysunek 2.16 Kształt deformacji słupka podporowego dla następujących rozmiarów oczek: (a) 5.0 mm; (b) 1.0 mm; (c) 1.0 mm (0.1 mm w pobliżu słupków wsporczych); oraz (d) 5.0 mm (0.5 mm w pobliżu podpór).

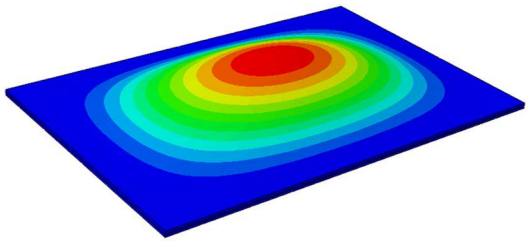
Wyznaczone częstotliwości drgań własnych to odpowiednio 5941.5, 6752.4, 7584.3 i 7634.6 Hz. Im mniejszy rozmiar siatki elementów skończonych tafli szkła w sąsiedztwie powierzchni styku, tym większe są w tych przypadkach omawiane częstotliwości drgań własnych. Dokładna analiza powyżej omówionego zagadnienia wraz z wynikami znajduje się w pracy [13].

Porównując uzyskane wyniki, stworzenie jednego malutkiego elementu skończonego pod słupkiem wsporczym jest najbardziej efektywnym rozwiązaniem (Rysunek 2.16 (d)), co prowadzi do dokładnego wyniku liczbowego i zmniejsza koszty obliczeniowe w porównaniu z założeniami przedstawionymi na sąsiednim rysunku ((Rysunek 2.16 (c)). Takie podejście przyjęto na potrzeby dalszej analizy.

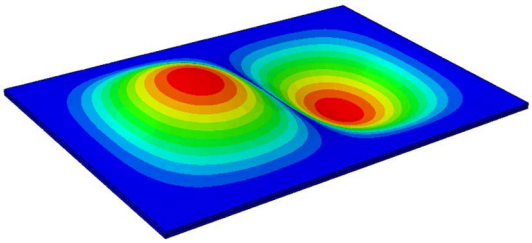
Na potrzeby dalszej analizy, jako maksymalny rozmiar oczka siatki elementów skończonych przyjęto 0.010 m. Jest to wystarczająco mały wymiar elementu, który pozwala uzyskać odpowiednio dokładny wynik przy możliwie najkrótszym czasie prowadzonych obliczeń.

2.3.1. Wyniki

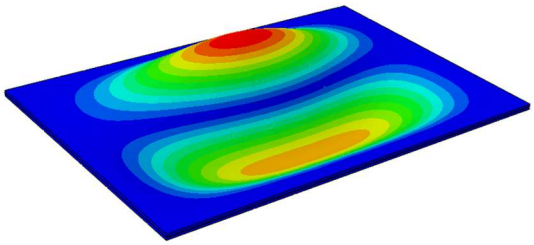
Dla modelu numerycznego uwzględniającego przyjęte w tym rozdziale założenia, wyznaczono trzydzieści pierwszych częstotliwości drgań własnych. Tablica 2.4 zawiera te wartości w kolumnie z wynikami dla modelu wykonanego w programie ABAQUS. Rysunek 2.17 ilustruje powiązane z tymi wartościami postacie drgań własnych.



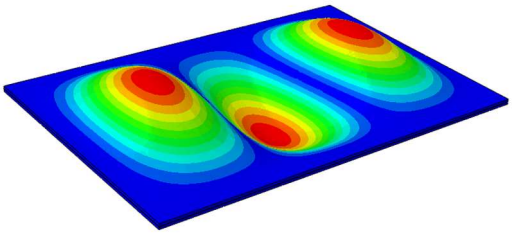
1



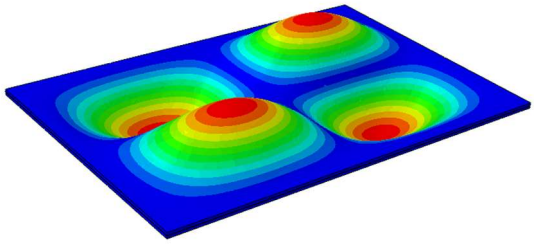
2



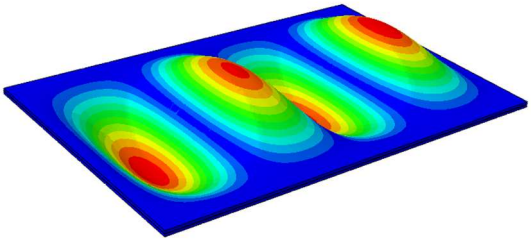
3



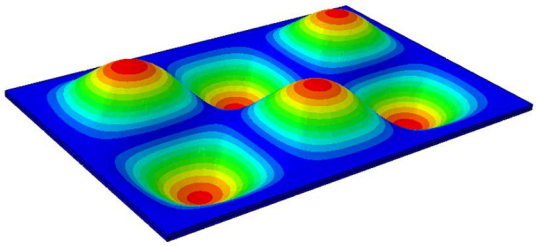
4



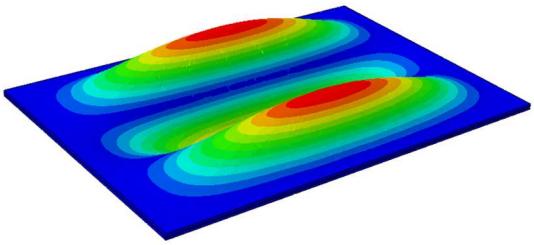
5



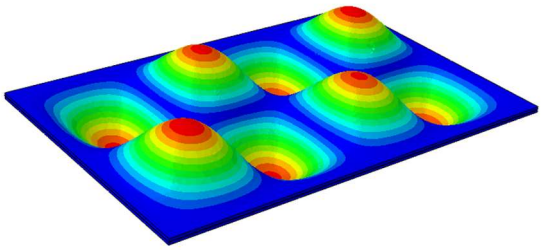
6



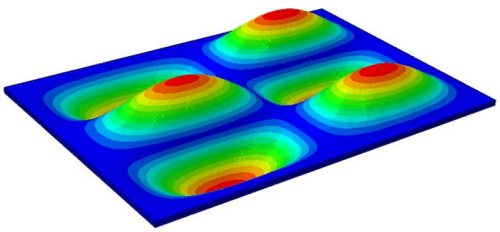
7



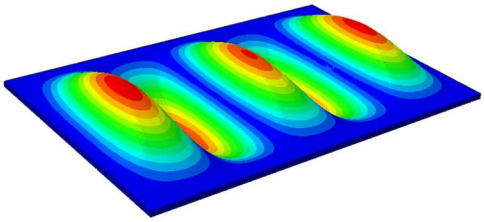
8



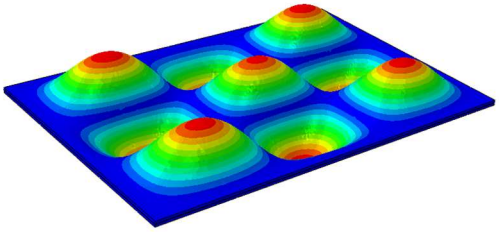
9



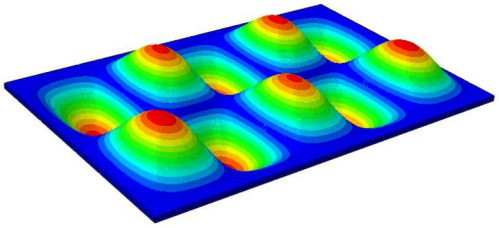
10



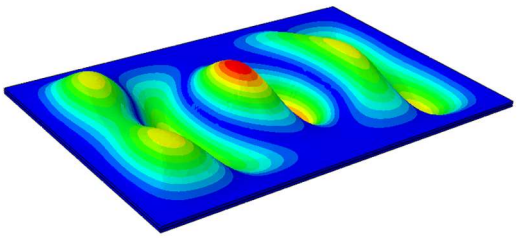
11



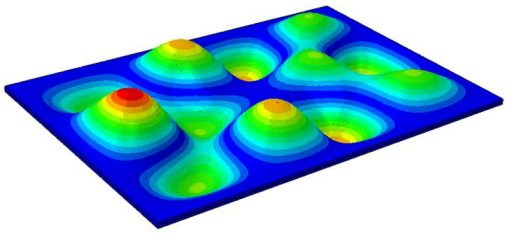
12



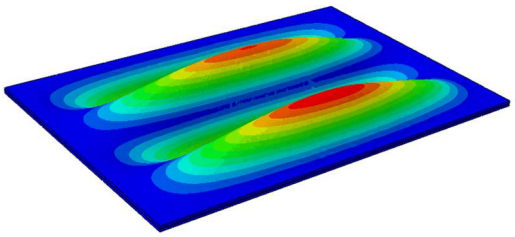
13



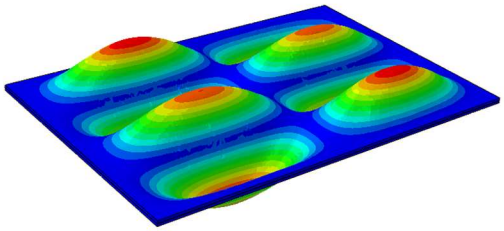
14



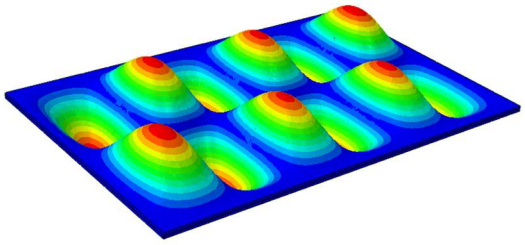
15



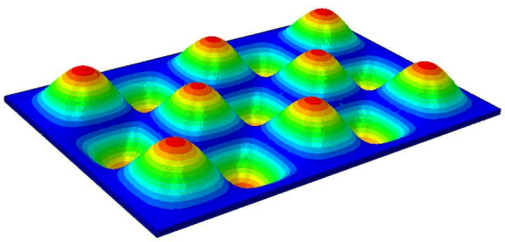
16



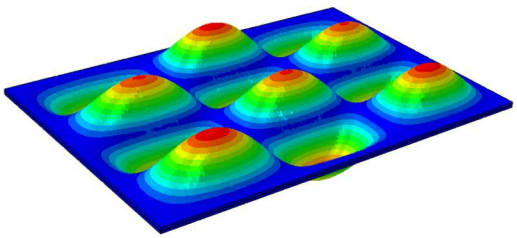
17



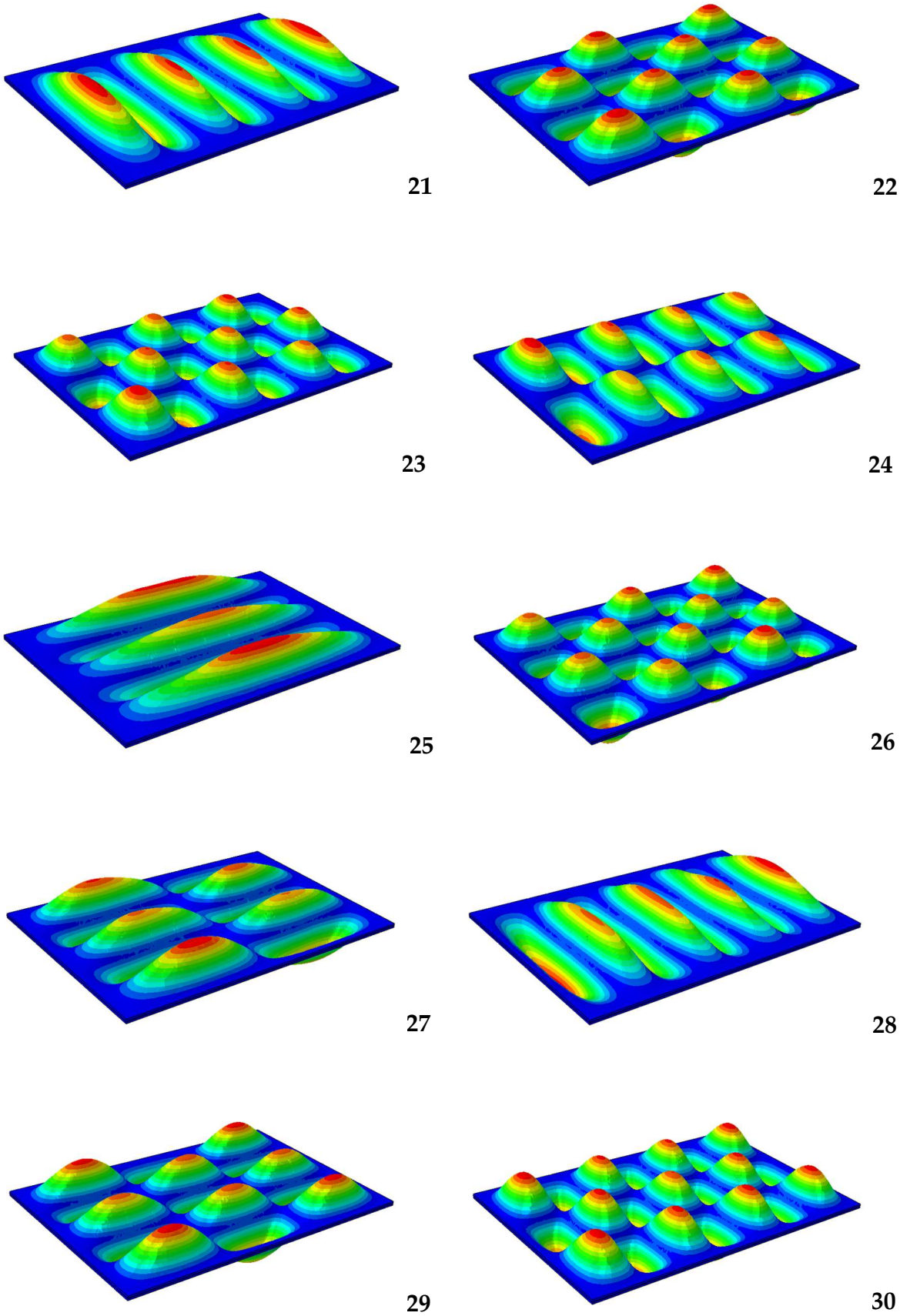
18



19



20



Rysunek 2.17 Postacie drgań własnych dla odpowiednio pierwszych trzydziestu częstotliwości drgań własnych – ABAQUS

2.3.2. Python jako akcelerator

W celu efektywnego tworzenia dużej ilości modeli numerycznych odwzorowujących płyty typu VIG, w których różne parametry mogą ulegać zmianie, wykorzystano ABAQUS Scripting w języku Python (wersja 3.9.5) jako alternatywę dla graficznego interfejsu użytkownika w programie ABAQUS. Stworzony skrypt umożliwia użytkownikowi określenie wszystkich właściwości zarówno geometrycznych jak i mechanicznych. Z uwagi na potrzebę wygenerowania wyników analizy dynamicznej dla dużej ilości kombinacji różnych parametrów, wykorzystanie tego skryptu znacząco skróciło proces generowania wyników.

Omówiony powyżej algorytm składa się z klasy, która opisuje cały proces tworzenia modelu numerycznego płyty VIG. Algorytm ten zawiera metody pozwalające na przypisanie wszystkich parametrów geometrycznych (Rysunek 2.1), wszystkich parametrów materiałowych oraz określenie rozmiaru siatki elementów skończonych, z uwzględnieniem również parametrów dogęszczania siatki na elementach odwzorowujących numerycznie tafle szkła (punkt 2.3.3). Główna metoda zaprogramowanej klasy jest odpowiedzialna nie tylko za stworzenie całego modelu numerycznego w oparciu o wartości określone w poprzednich metodach, ale również za ustalenie wybranych warunków brzegowych oraz określenie ilości poszukiwanych wartości własnych. Ponadto pozwala on również na uruchomienie obliczeń oraz zapisanie uzyskanych wyników do zewnętrznego pliku tekstowego.

Pomimo tego, że na potrzeby opisanych w dalszych rozdziałach analiz, jako zmienne przyjęto tylko wybrane parametry, skrypt pozwala również na zmianę wszystkich pozostałych parametrów.

Kod źródłowy algorytmu opisanego w tym punkcie znajduje się w załączniku nr 1.

2.4. Porównanie otrzymanych wyników

Porównano ze sobą, otrzymane na podstawie modeli numerycznych stworzonych przy użyciu oprogramowania RFEM oraz ABAQUS, wartości częstotliwości drgań własnych płyt VIG. Celem odpowiedniego odwzorowania numerycznego stalowych pilastrów w modelu powłokowo-prętowym (RFEM), uwzględniono ich skończoną sztywność obrotową w węzłach.

Słupki podporowe, oddzielające od siebie dwie tafle szkła, są w przybliżeniu walcami o średnicy podstawy 0,6 mm oraz wysokości 0,3 mm. Można zatem zaklasyfikować je do elementów krępych. Mimo to, nie można jednoznacznie określić, czy przyjęcie nieskończonej sztywności obrotowej w węźle łączącym prętowy element skończony, odwzorowujący numerycznie słupek, wraz z powłokowym elementem skończonym, odwzorowującym numerycznie taflę szkła, jest zabiegiem poprawnym. W celu porównania wyników otrzymanych przy uwzględnieniu różnych sposobów odwzorowania tego połączenia w modelu powłokowo-prętowym (RFEM) z wynikami otrzymanymi przy użyciu modelu trójwymiarowego (ABAQUS), stworzono trzy różne modele powłokowo-prętowe uwzględniające następujące założenia dotyczące omawianej sztywności obrotowej węzła: połączenie przegubowe (RFEM_1), połączenie o skończonej sztywności obrotowej o wartości 0.4 Nm/rad (RFEM_2) oraz połączenie sztywne (RFEM_3). Tablica 2.4 zawiera opisane powyżej porównanie wyników otrzymanych przy użyciu tych modeli.

Tablica 2.4 Wyniki otrzymane dla różnych sposobów numerycznego odwzorowania uszczelki

Nr częstotliwości drgań własnych	Częstotliwość dla danego modelu [Hz]				Różnica pomiędzy danym modelem z RFEM, a modelem z ABAQUS		
	RFEM_1	RFEM_2	RFEM_3	ABAQUS	RFEM_1	RFEM_2	RFEM_3
1	200.05	190.27	180.06	202.70	-12.58%	-6.53%	-1.32%
2	300.99	288.60	275.96	299.06	-8.37%	-3.62%	0.64%
3	467.24	452.76	438.56	452.83	-3.25%	-0.02%	3.18%
4	468.18	455.15	442.24	453.56	-2.56%	0.35%	3.22%
5	558.90	544.79	531.08	537.23	-1.16%	1.41%	4.03%
6	694.93	679.04	663.85	658.33	0.84%	3.15%	5.56%
7	712.54	697.28	682.72	678.09	0.68%	2.83%	5.08%
8	867.96	853.42	839.64	812.51	3.34%	5.04%	6.82%
9	929.31	913.11	897.86	872.93	2.86%	4.60%	6.46%
10	954.62	939.56	925.35	893.21	3.60%	5.19%	6.87%
11	981.99	965.17	949.34	913.07	3.97%	5.71%	7.55%
12	1095.80	1085.30	1070.58	1028.6	4.08%	5.51%	6.53%

13	1101.00	1095.80	1095.80	1120.5	-2.25%	-2.25%	-1.77%
14	1199.84	1190.65	1174.86	1216.5	-3.54%	-2.17%	-1.39%
15	1207.57	1199.84	1199.84	1216.7	-1.41%	-1.41%	-0.76%
16	1308.69	1292.33	1277.10	1280.7	-0.28%	0.91%	2.19%
17	1329.09	1311.67	1295.40	1360.3	-5.01%	-3.71%	-2.35%
18	1400.47	1385.49	1371.38	1418.5	-3.44%	-2.38%	-1.29%
19	1485.21	1470.00	1455.78	1459.6	-0.26%	0.71%	1.75%
20	1521.22	1521.22	1514.23	1492.9	1.43%	1.90%	1.90%

Należy jednak zauważyć, że we wspomnianych wcześniej modelach powłokowo-prętowych, rozpatrywane charakterystyki dotyczące sztywności obrotowej węzłów końcowych elementów prętowych, odwzorowujących numerycznie pilastry, były zakładane jako jednakowe w obrębie całego danego modelu. Oznacza to, że w jednym modelu wszystkie te połączenia były przegubowe (RFEM_1), w innym podatne (RFEM_2), a w jeszcze innym sztywne (RFEM_3). Takie podejście może generować niepożądane oraz trudne do określenia błędy numerycznego odwzorowania rzeczywistego elementu. Sztywność obrotową węzła, poza elementami łączonymi, definiuje również stan odkształceń panujący w tym połączeniu. Określenie poziomu odkształceń w każdym z połączeń pilastrów z taflami szkła w modelu powłokowo-prętowym, a następnie iteracyjne określenie wartości sztywności obrotowej takiego połączenia jest zagadnieniem niezwykle skomplikowanym. Powoduje to komplikacje nie tylko całego zagadnienia, ale również modelu numerycznego, co znacząco wydłuża czas obliczeń, przy czym nadal może prowadzić do otrzymywania nieadekwatnych wyników.

Innym istotnym aspektem problematyki numerycznego odwzorowania omawianego zagadnienia w modelu powłokowo-prętowym jest różnica odległości powierzchni środkowych dwóch tafli szkła, odniesiona da różnicy odległości sąsiadujących ze sobą powierzchni tych elementów. Dla omawianego w tym rozdziale elementu jest to stosunek określony poniższym wyrażeniem (2.6):

$$\frac{2 \cdot 0.5 \cdot t_g + t_v}{t_v} = \frac{4{,}3\ mm}{0{,}3\ mm} = 14{,}33\,. \tag{2.6}$$

Wartość ta generuje problemy wynikające nie tylko ze sposobu numerycznego odwzorowania stalowych pilastrów, ale również z omówionego wcześniej sposobu numerycznego odwzorowania uszczelki.

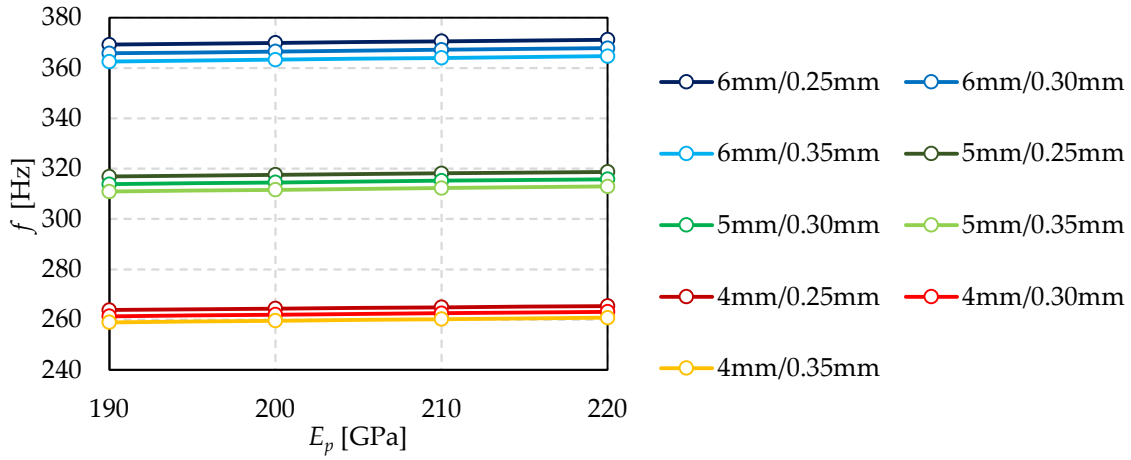
Biorąc pod uwagę powyższe, stworzone modele powłokowo-prętowe wykorzystano jako modele do walidacji, przyjętego do dalszych analiz modelu trójwymiarowego, wykonanego w oprogramowaniu ABAQUS.

3. Analiza dynamicznego zachowania płyt typu VIG

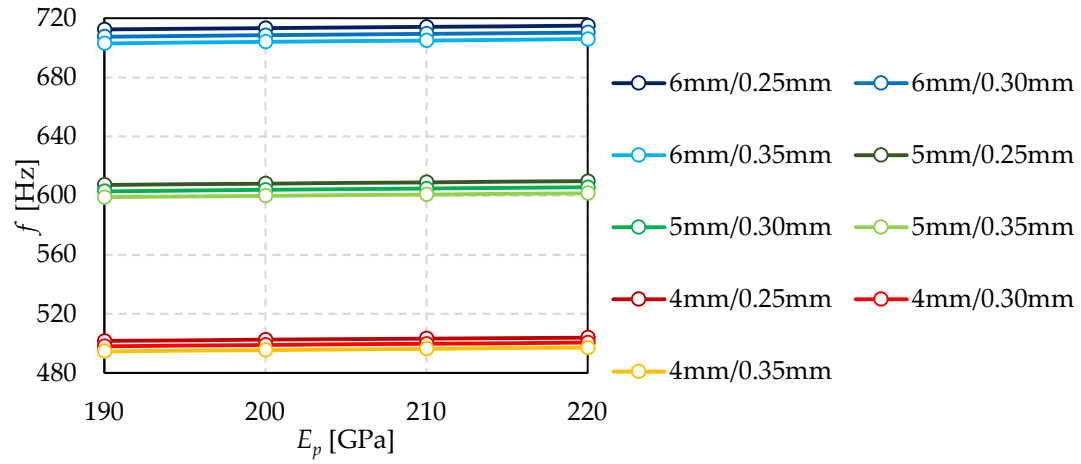
3.1. Wpływ modułu sprężystości podłużnej słupków na częstotliwość drgań własnych płyty

W poniższym punkcie zestawiono wykresy zależności wartości częstotliwości drgań własnych dla pierwszej oraz drugiej postaci drgań własnych od modułu sprężystości podłużnej stalowych pilastrów dla różnych rozpatrywanych kombinacji grubości tafli szkła oraz wysokości warstwy próżni. Wykresy te wykonano dla każdej z dziesięciu kombinacji wymiarów bocznych płyt VIG.

A. Płyta VIG o wymiarach 0.4 m x 0.4 m

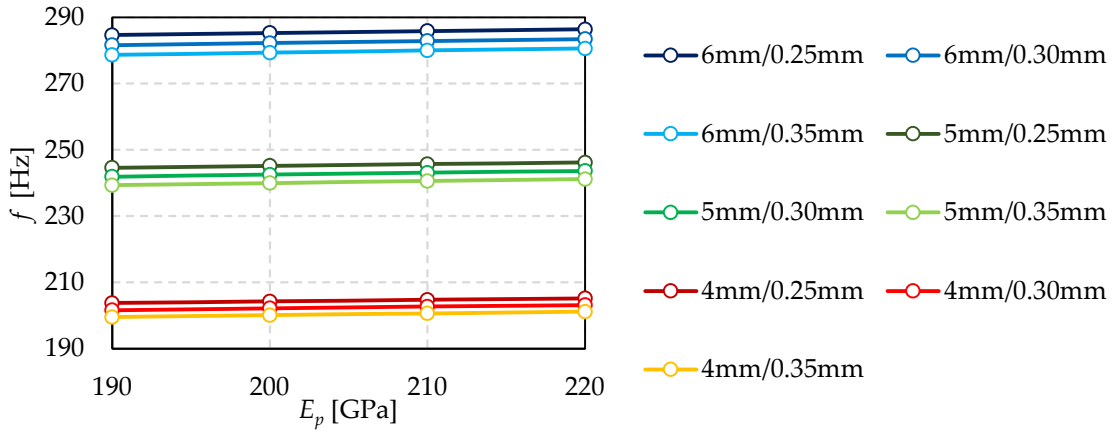


Rysunek 3.1 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 0.4 m

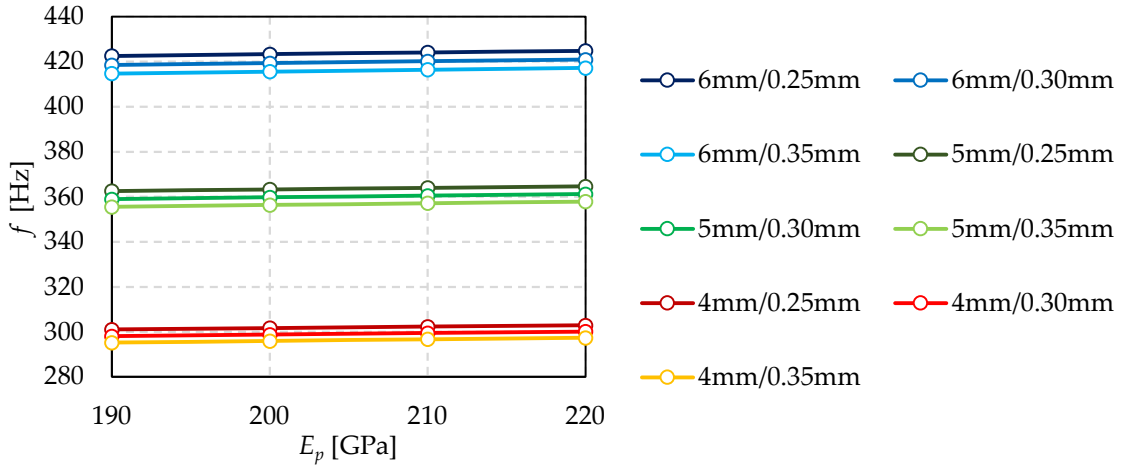


Rysunek 3.2 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 0.4 m

B. Płyta VIG o wymiarach 0.4 m x 0.6 m

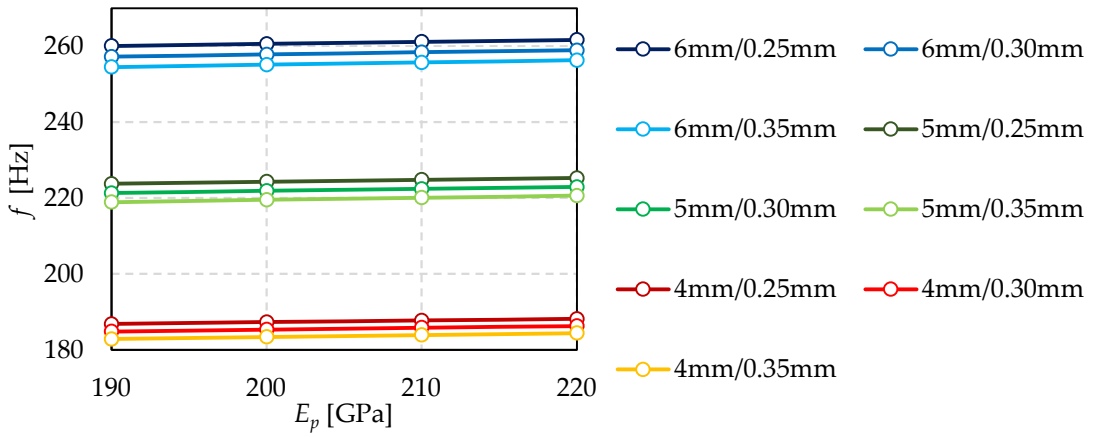


Rysunek 3.3 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 0.6 m

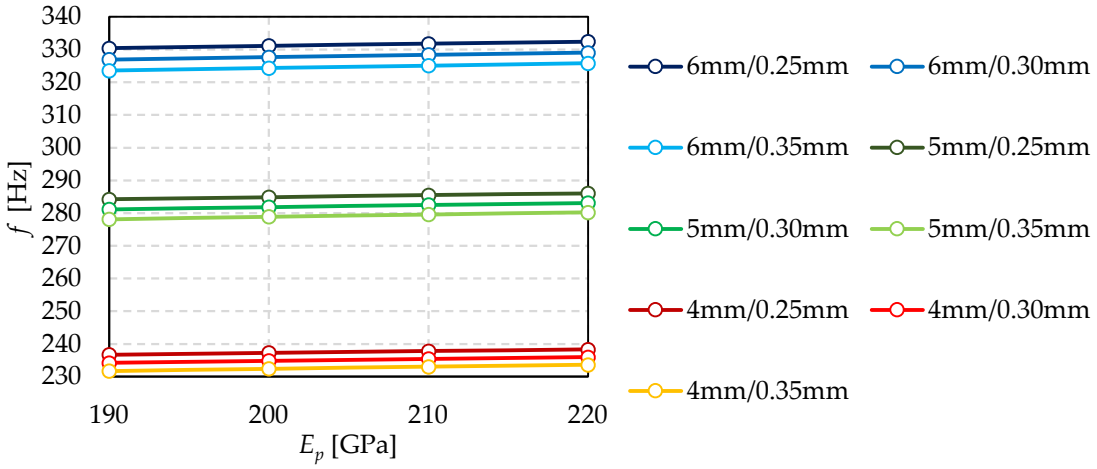


Rysunek 3.4 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 0.6 m

C. Płyta VIG o wymiarach 0.4 m x 0.8 m

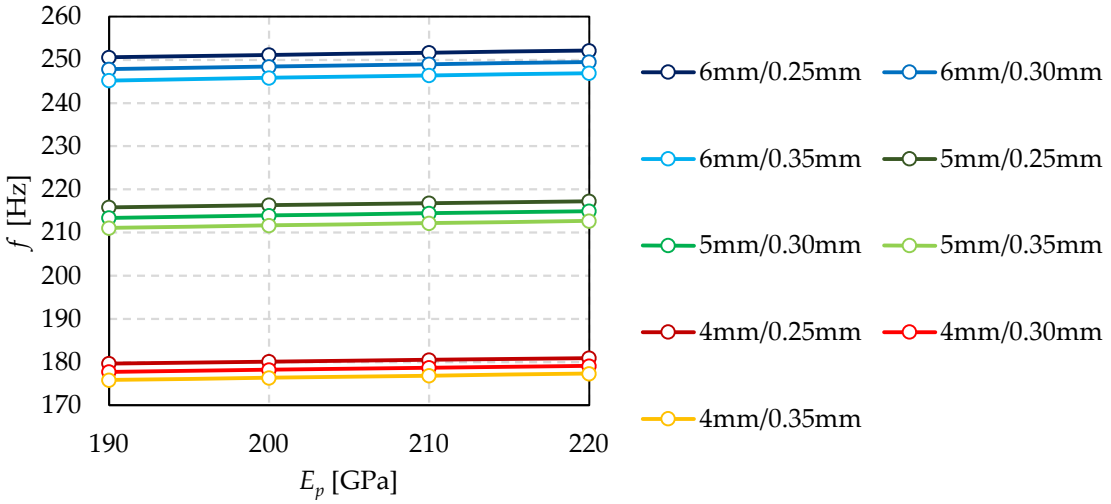


Rysunek 3.5 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 0.8 m

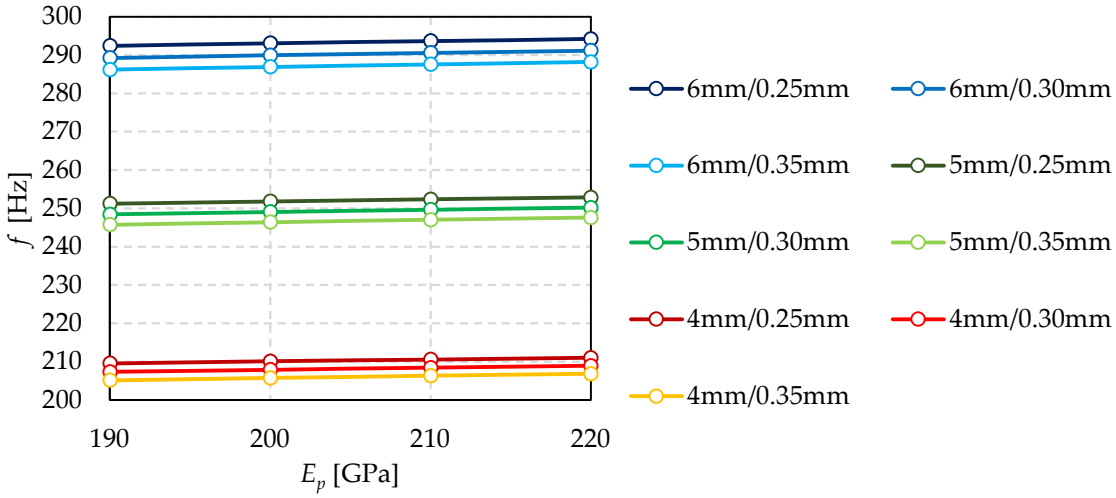


Rysunek 3.6 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 0.8 m

D. Płyta VIG o wymiarach 0.4 m x 1.0 m

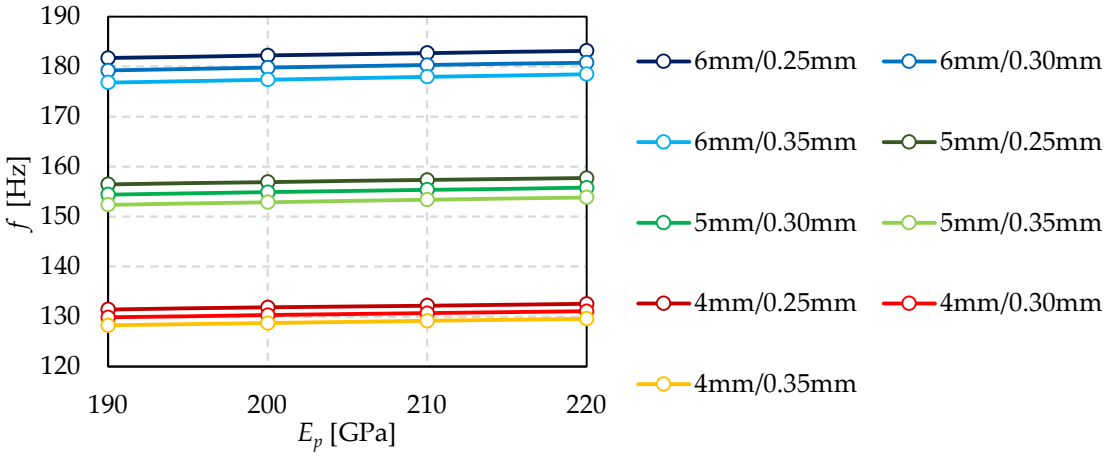


Rysunek 3.7 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 1.0 m

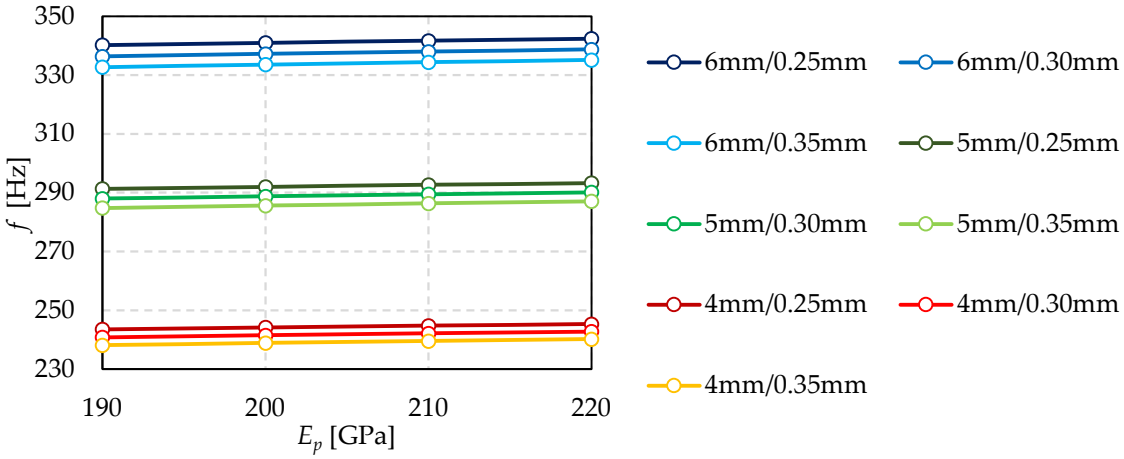


Rysunek 3.8 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.4 m x 1.0 m

E. Płyta VIG o wymiarach 0.6 m x 0.6 m

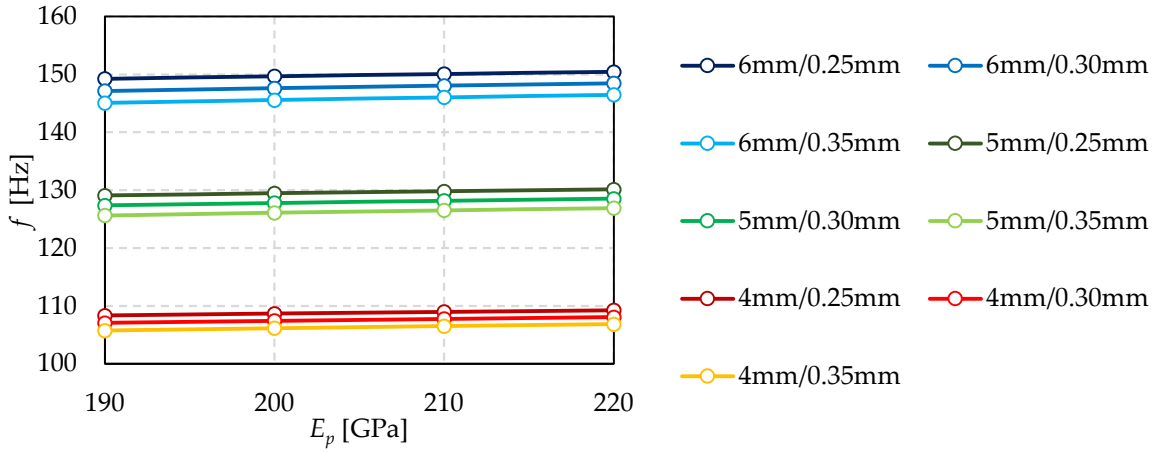


Rysunek 3.9 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.6 m x 0.6 m

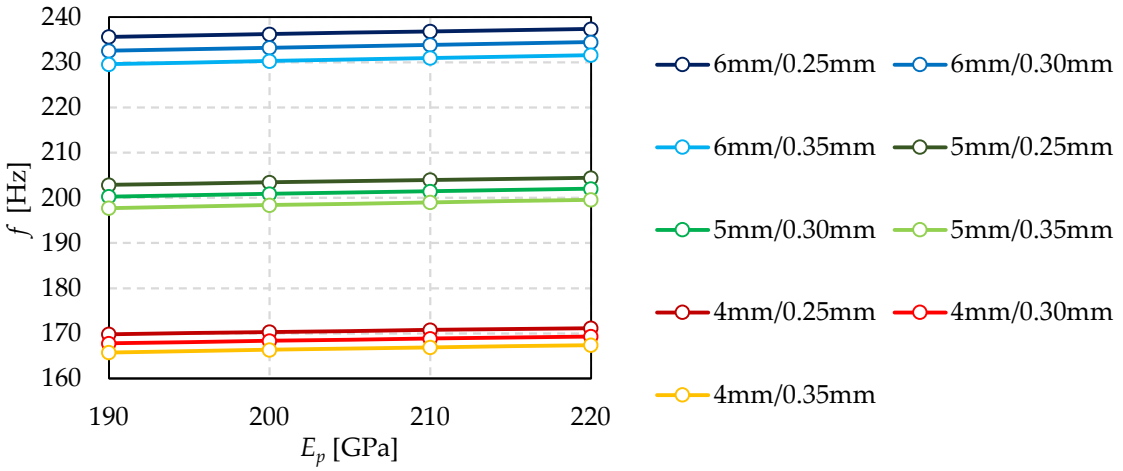


Rysunek 3.10 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.6 m x 0.6 m

F. Płyta VIG o wymiarach 0.6 m x 0.8 m

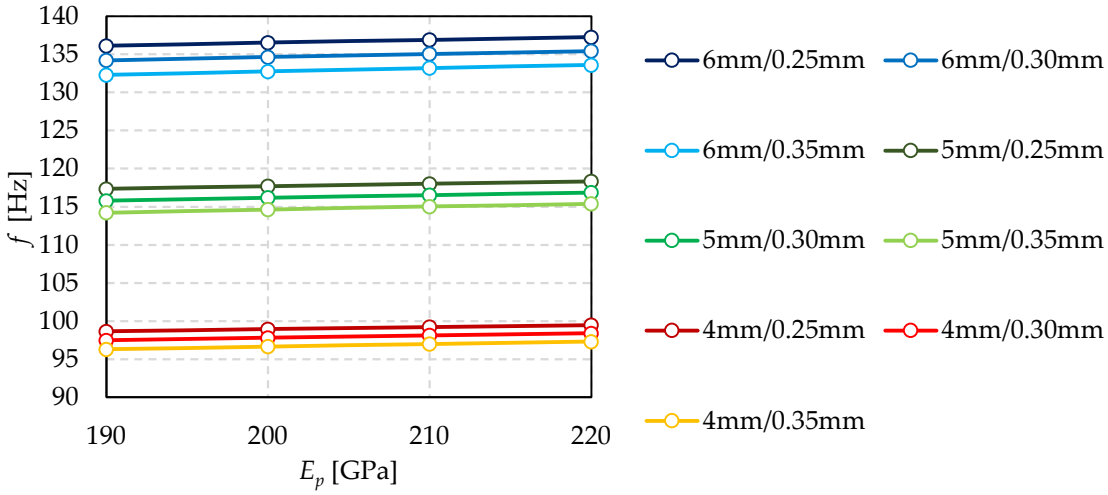


Rysunek 3.11 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.6 m x 0.8 m

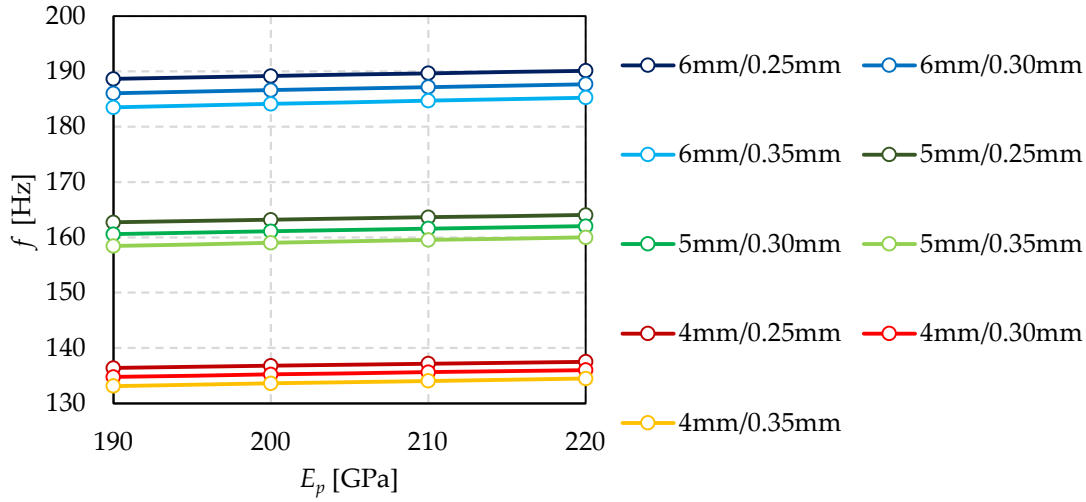


Rysunek 3.12 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.6 m x 0.8 m

G. Płyta VIG o wymiarach 0.6 m x 1.0 m

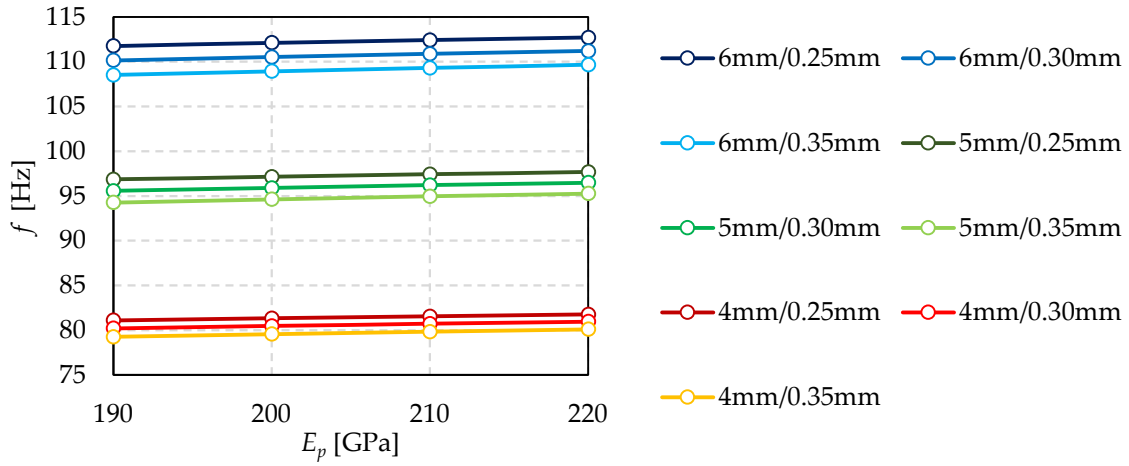


Rysunek 3.13 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.6 m x 1.0 m

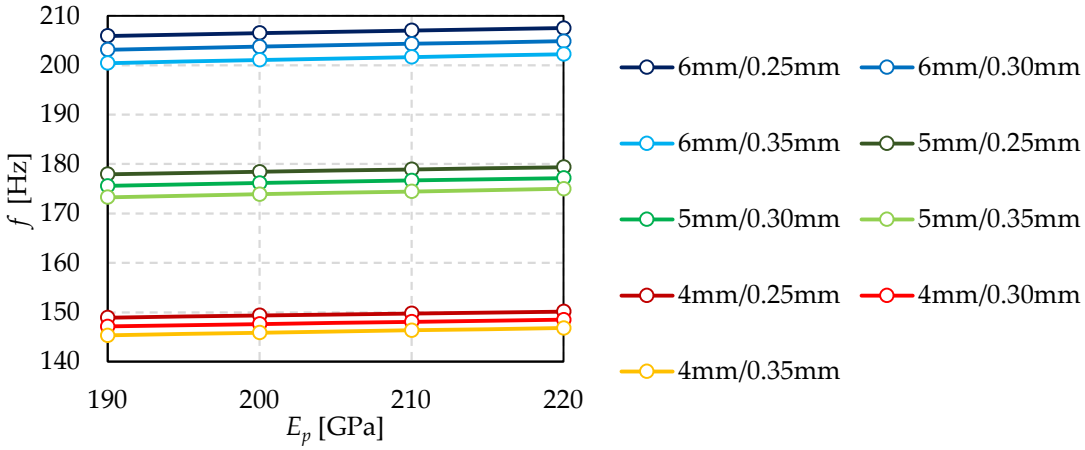


Rysunek 3.14 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.6 m x 1.0 m

H. Płyta VIG o wymiarach 0.8 m x 0.8 m

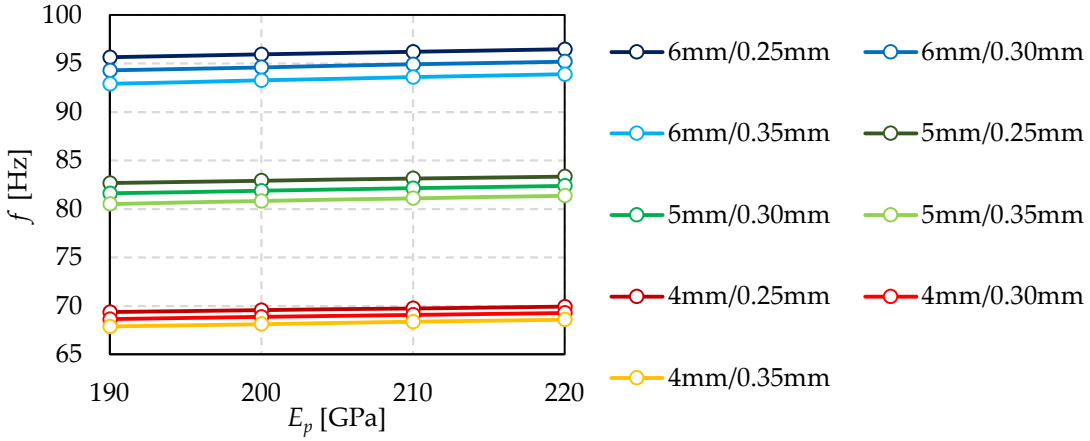


Rysunek 3.15 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.8 m x 0.8 m

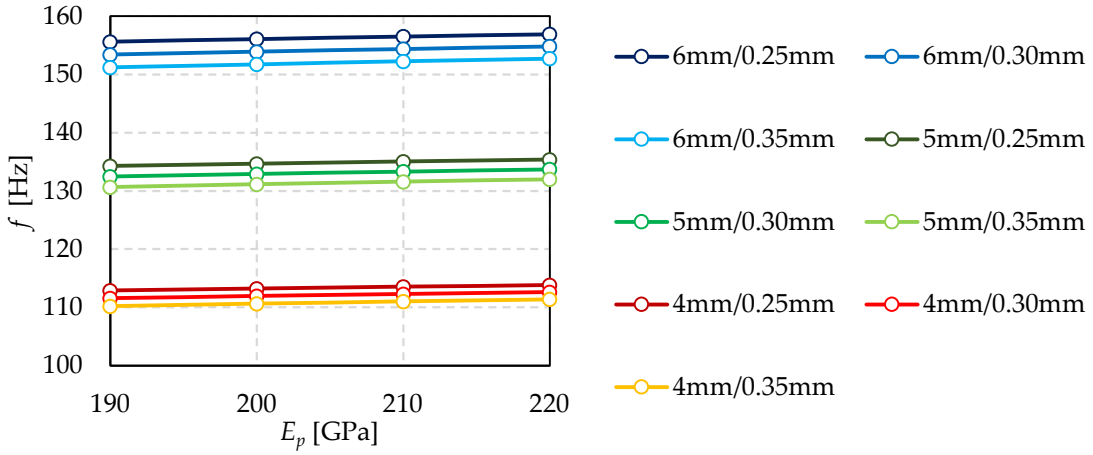


Rysunek 3.16 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.8 m x 0.8 m

I. Płyta VIG o wymiarach 0.8 m x 1.0 m

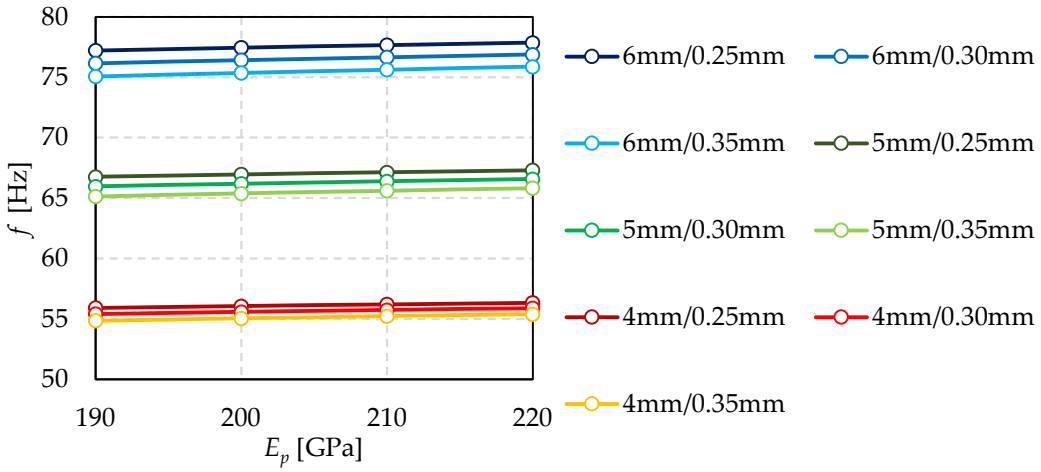


Rysunek 3.17 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.8 m x 1.0 m

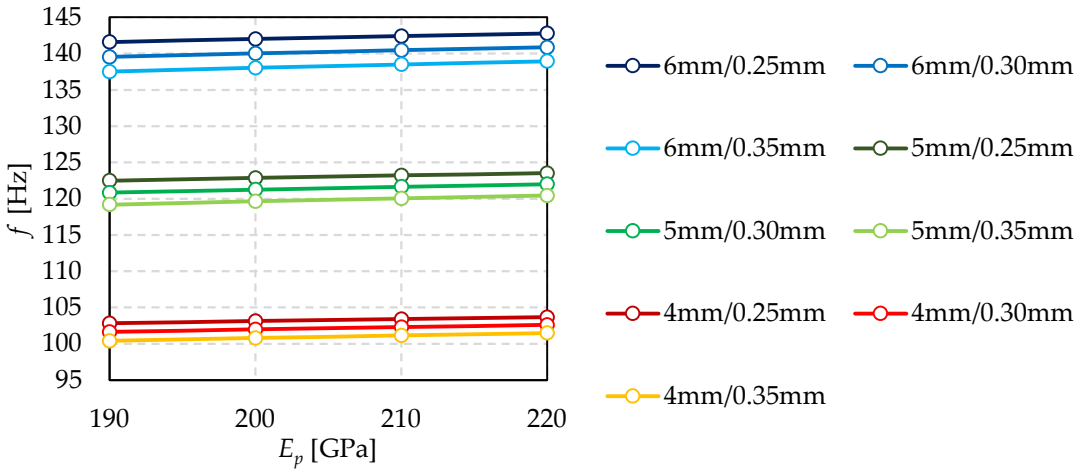


Rysunek 3.18 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 0.8 m x 1.0 m

J. Płyta VIG o wymiarach 1.0 m x 1.0 m



Rysunek 3.19 Pierwsza częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 1.0 m x 1.0 m



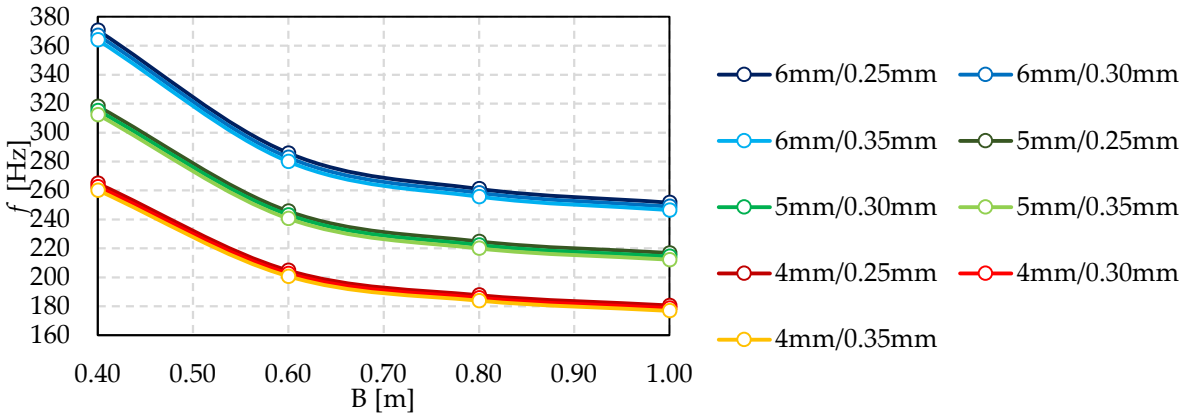
Rysunek 3.20 Druga częstotliwość drgań własnych płyty w zależności od modułu Younga pilastrów dla różnych grubości szkła i wysokości próżni – 1.0 m x 1.0 m

Na podstawie przedstawionych w tym punkcie wykresów można stwierdzić, że wpływ moduł sprężystości podłużnej pilastrów oraz wysokości warstwy próżni na wartość częstotliwości drgań własnych płyt typu VIG jest niewielki w porównaniu do wpływu grubości tafli szkła na tę wartość. Należy również zauważyć, że im większa grubość tafli szkła, tym bardziej wyraźny staje się wpływ wysokości próżni na wartości częstotliwości drgań własnych szyb próżniowych.

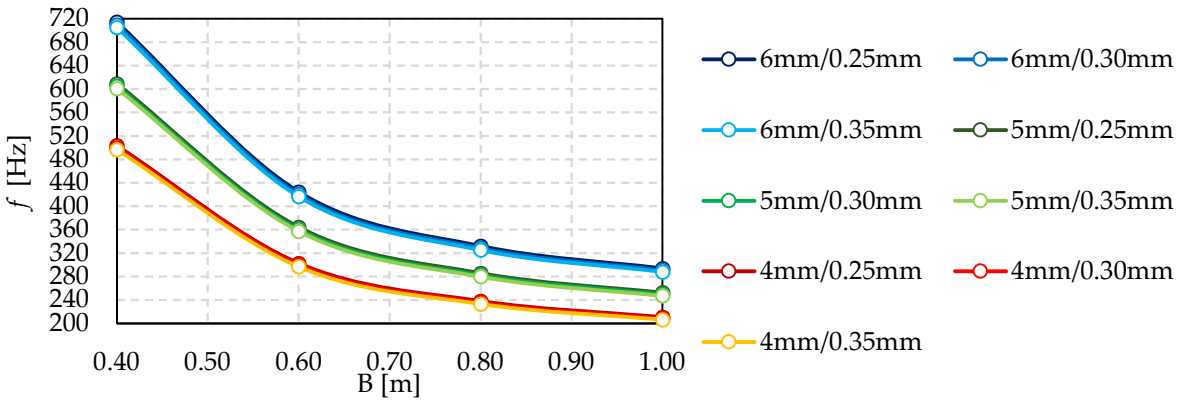
3.2. Wpływ wymiarów płyty na jej częstotliwość drgań własnych

W poniższym punkcie przedstawiono za pomocą wykresów zależność wartości częstotliwości drgań własnych od długości jednego z boków kwadratowej płyty VIG (przy ustalonej długości drugiego z jej boków). Wykresy sporządzono dla pierwszych czterech częstotliwości drgań własnych przy ustalonej wartości modułu sprężystości podłużnej pilastrów, wynoszącej 210 GPa.

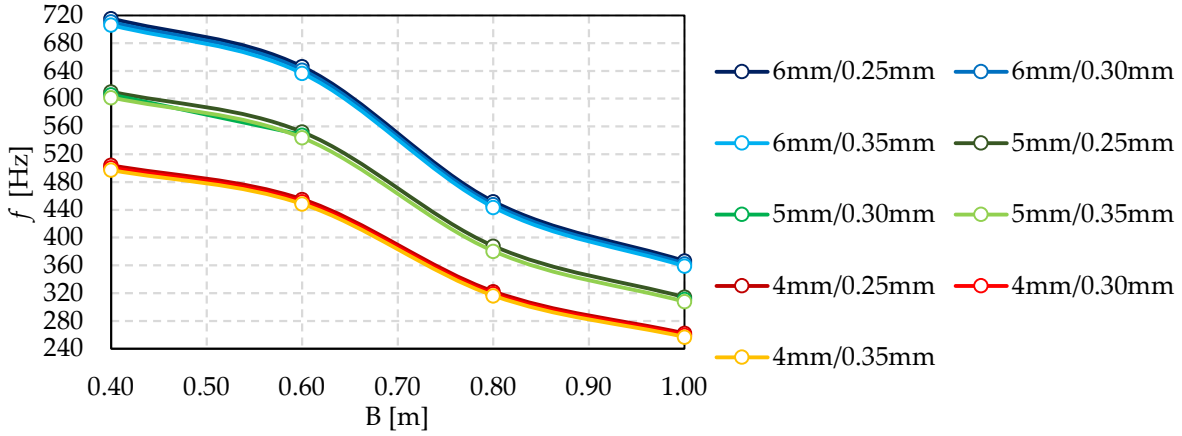
A. Jeden z boków o wymiarze 0.4 m



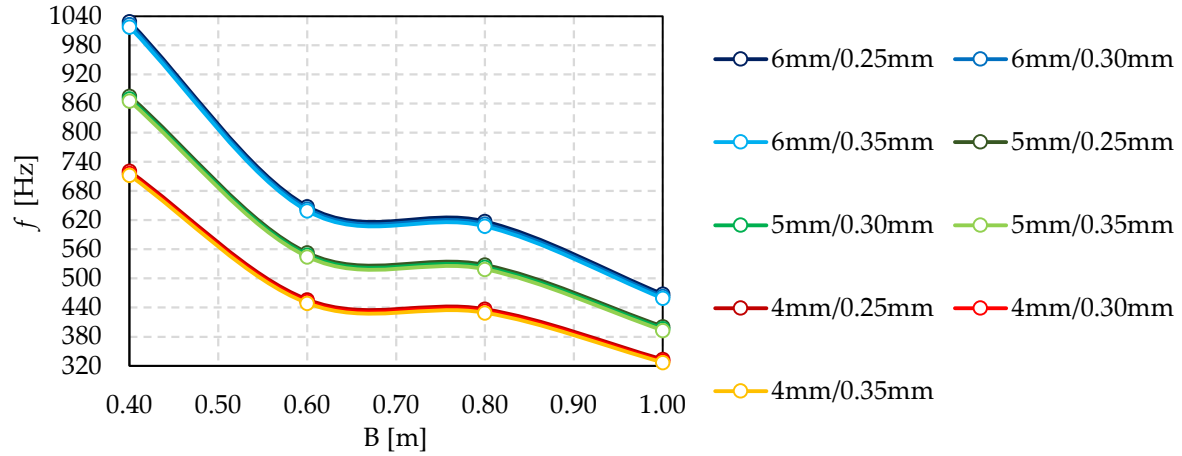
Rysunek 3.21 Pierwsza częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.4 m



Rysunek 3.22 Druga częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.4 m

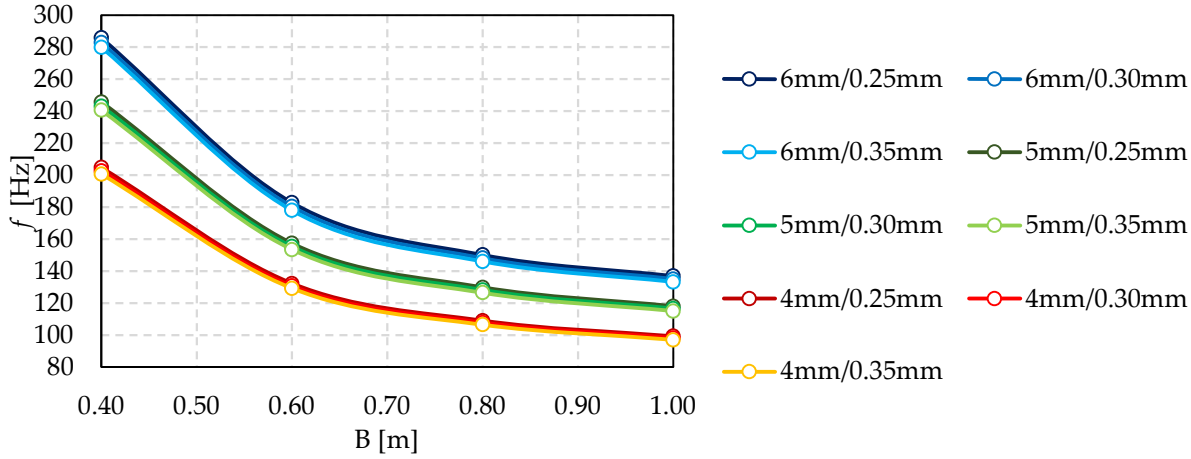


Rysunek 3.23 Trzecia częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.4 m

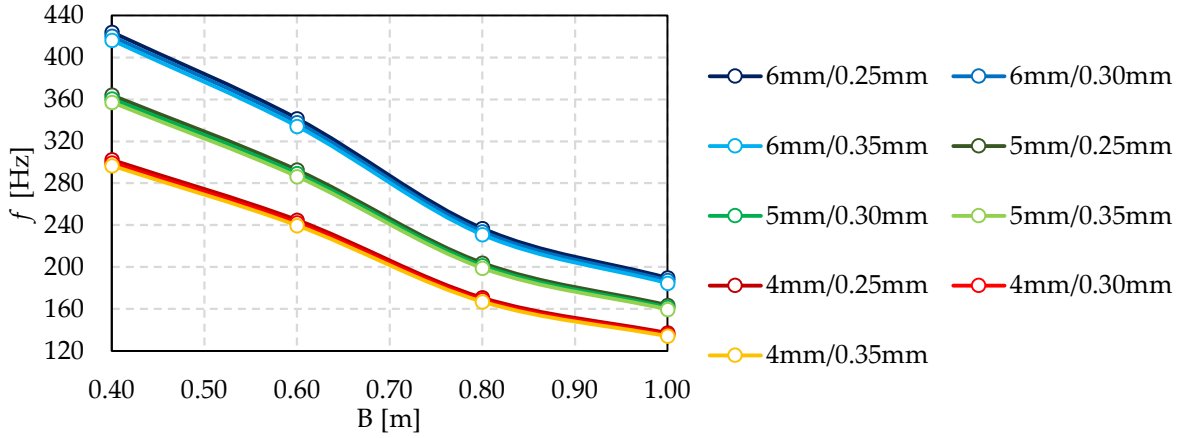


Rysunek 3.24 Czwarta częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.4 m

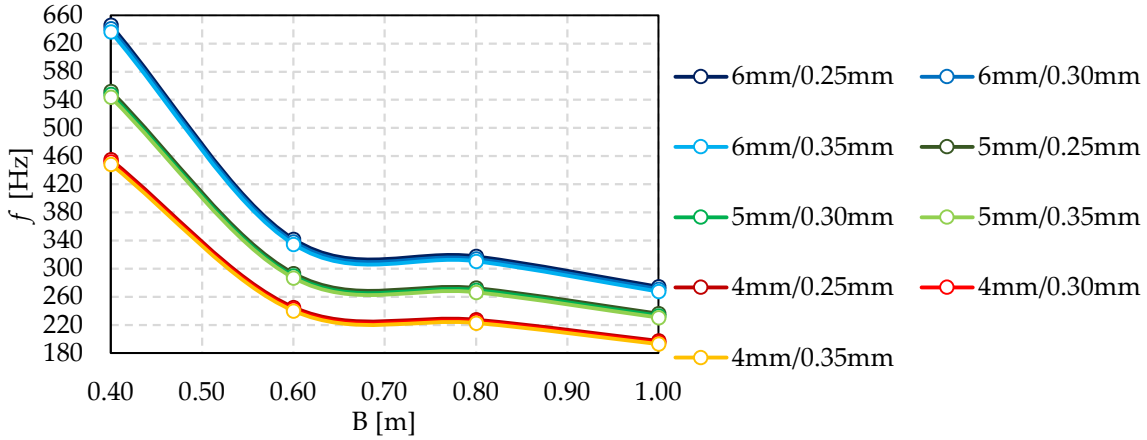
B. Jeden z boków o wymiarze 0.6 m



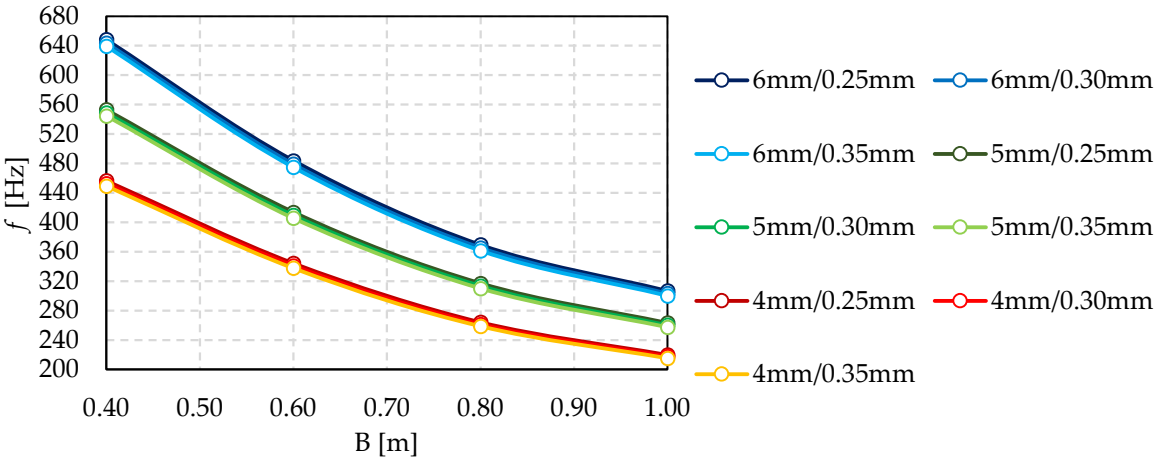
Rysunek 3.25 Pierwsza częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.6 m



Rysunek 3.26 Druga częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.6 m

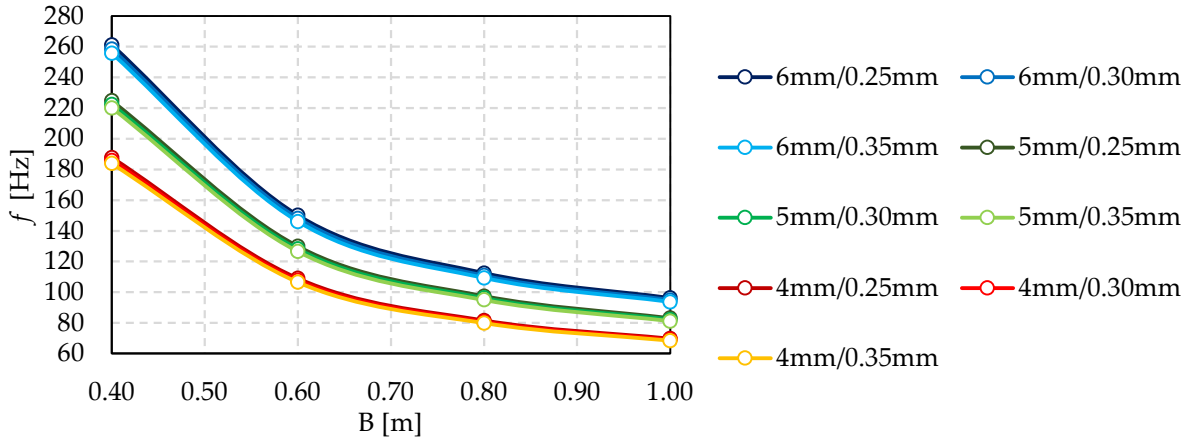


Rysunek 3.27 Trzecia częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.6 m

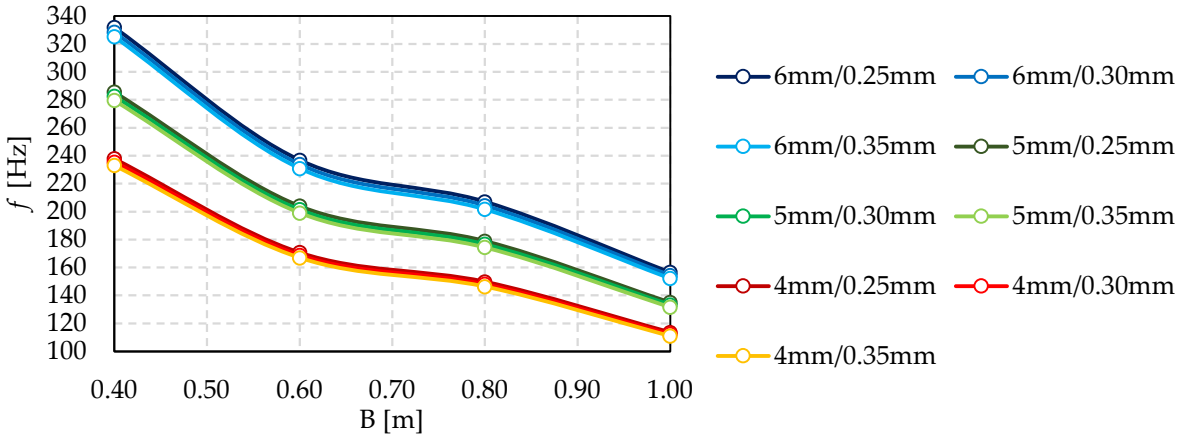


Rysunek 3.28 Czwarta częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.6 m

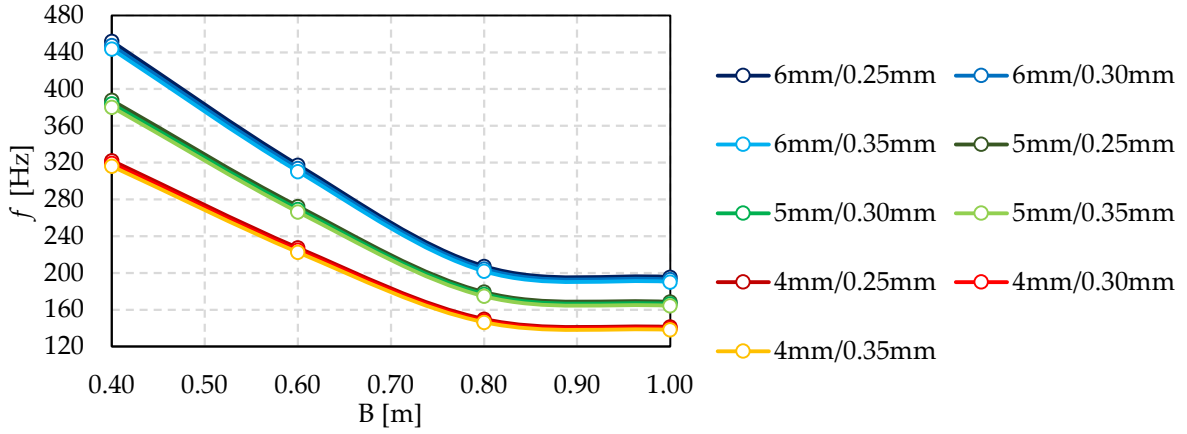
C. Jeden z boków o wymiarze 0.8 m



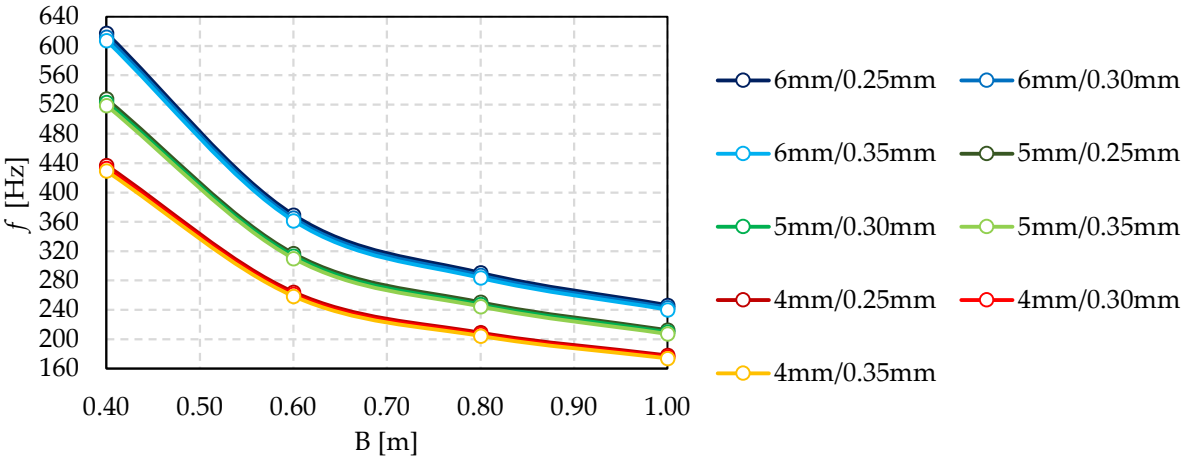
Rysunek 3.29 Pierwsza częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.8 m



Rysunek 3.30 Druga częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.8 m

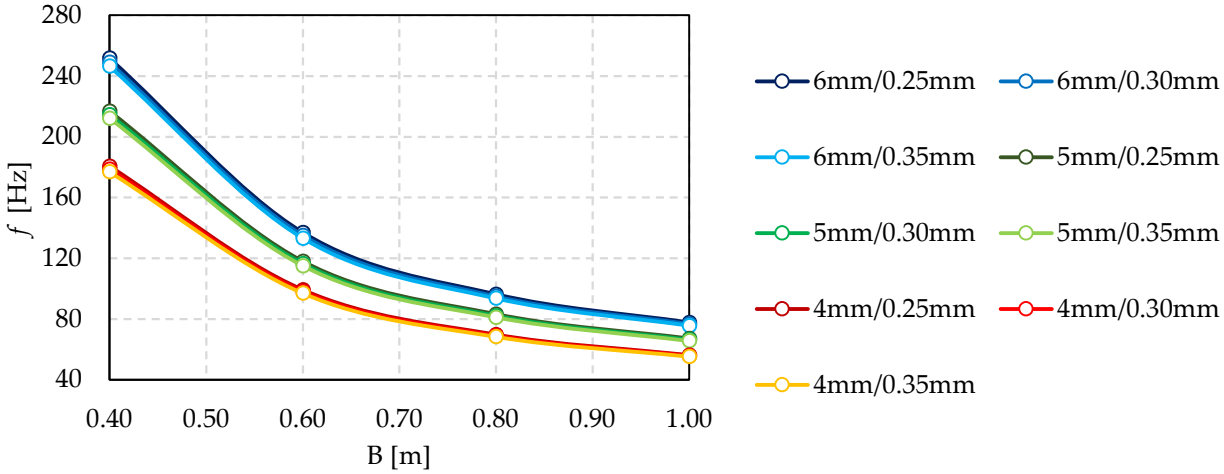


Rysunek 3.31 Trzecia częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.8 m

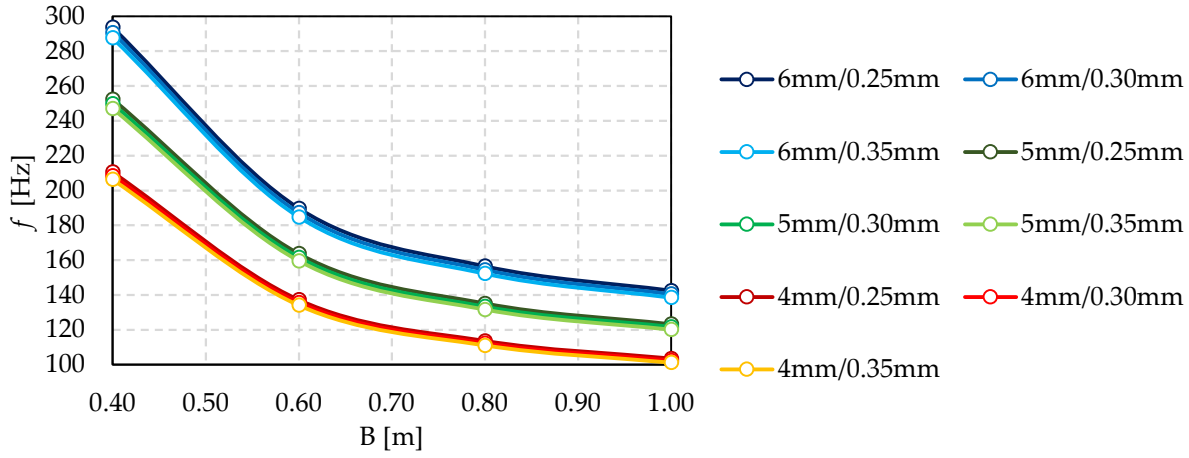


Rysunek 3.32 Czwarta częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 0.8 m

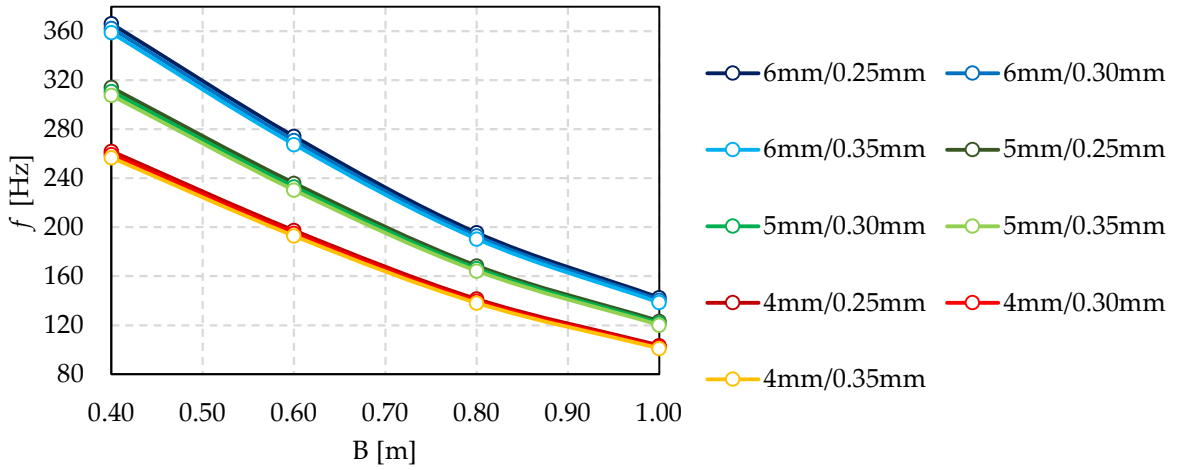
D. Jeden z boków o wymiarze 1.0 m



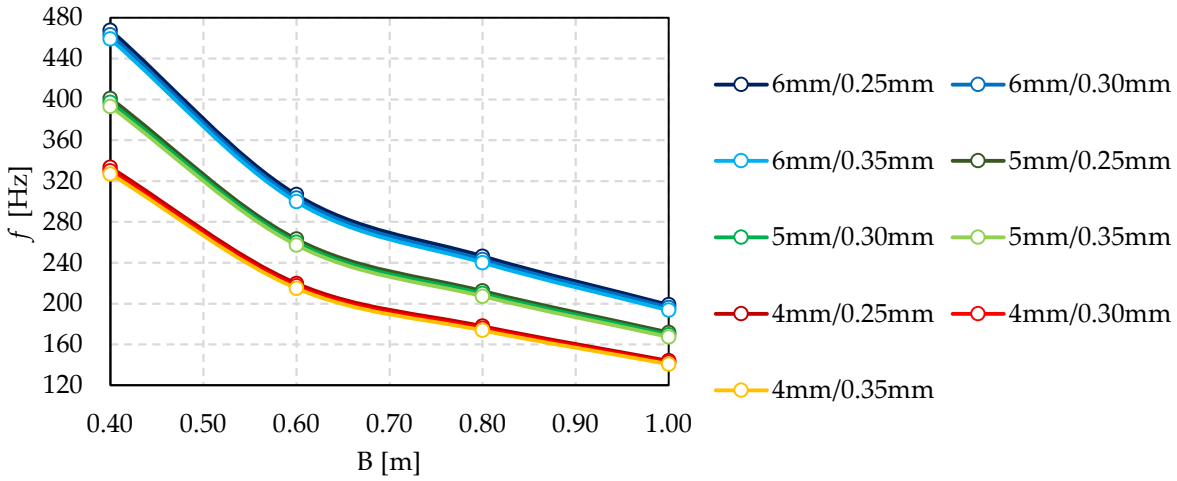
Rysunek 3.33 Pierwsza częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 1.0 m



Rysunek 3.34 Druga częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 1.0 m



Rysunek 3.35 Trzecia częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 1.0 m



Rysunek 3.36 Czwarta częstotliwość drgań własnych płyty w zależności od długości jednego z boków płyty dla różnych grubości szkła i wysokości próżni – drugi bok o długości 1.0 m

Na podstawie wykresów przedstawionych w tym punkcie można stwierdzić, że wpływ długości jednego z boków na pierwszą częstotliwość drgań własnych jest istotny, gdy stosunek jego długości do długości drugiego z boków jest mniejszy lub równy około 2.00. Ponadto, można zauważyć, że dla trzeciej oraz czwartej częstotliwości drgań własnych omawiana zależność ma inny charakter – można wyodrębnić punkty przegięcia funkcji przybliżającej tę zależność.

Należy również zauważyć, że niezależnie od grubości tafli szkła oraz wysokości próżni, charakter zmiany częstotliwości drgań własnych z uwagi na długość jednego z boków jest jednakowy.

3.3. Wpływ grubości tafli szkła oraz wysokości próżni na wzajemną relację pierwszych dziesięciu częstotliwości drgań własnych płyty

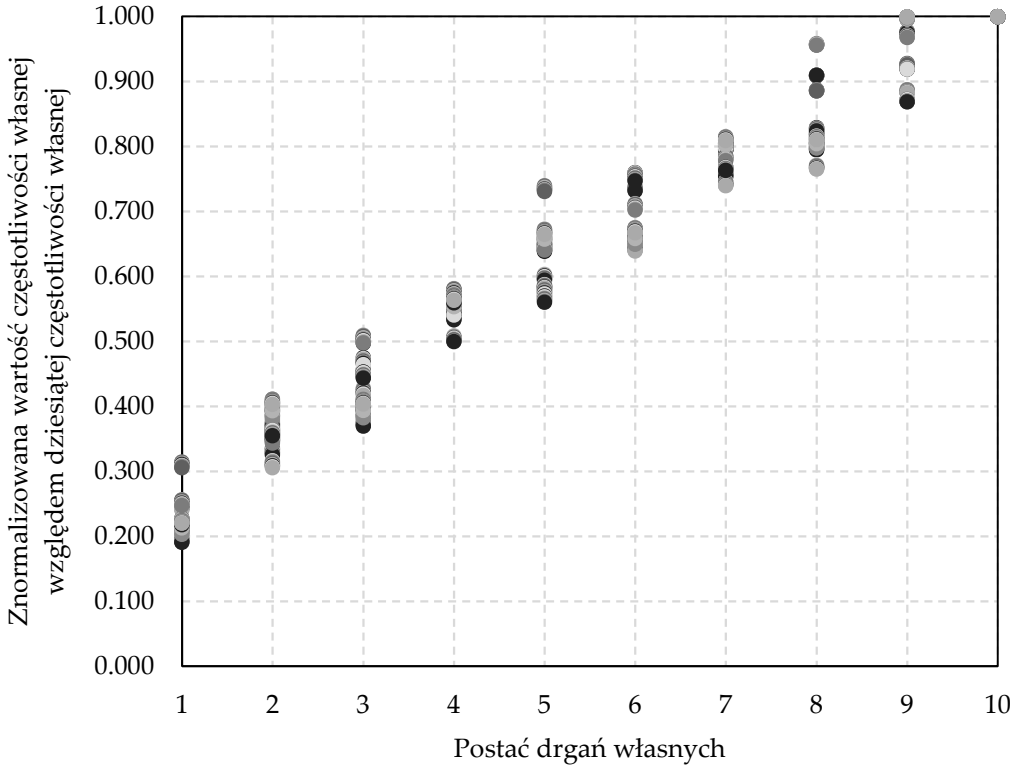
Tę część bieżącego rozdziału poświęcono określeniu wpływu grubości tafli szkła oraz wysokości próżni w płycie typu VIG na wzajemną relację pierwszych dziesięciu częstotliwości drgań własnych płyty. Wyznaczono znormalizowane wartości pierwszych dziesięciu częstotliwości drgań własnych w odniesieniu do dziesiątej częstotliwości drgań własnych. Przyjęta wartość modułu sprężystości podłużnej pilastrów wynosi 210 GPa. Tablica 3.1 przedstawia zestawienie opisanych powyżej wartości znormalizowanych dla wszystkich rozważnych wymiarów płyt. Rysunek 3.37 ilustruje te wartości.

Tablica 3.1 Zestawienie znormalizowanych wartości pierwszych dziesięciu częstotliwości własnych względem dziesiątej częstotliwości własnej

A [m]	B [m]	t_g [mm]	t_v [mm]	1	2	3	4	5	6	7	8	9	10
0.4	0.4	4.0	0.25	0.200	0.380	0.380	0.544	0.646	0.649	0.802	0.803	0.999	1.000
			0.30	0.199	0.378	0.379	0.542	0.645	0.648	0.801	0.802	0.999	1.000
			0.35	0.198	0.377	0.378	0.541	0.644	0.647	0.800	0.801	0.999	1.000
		5.0	0.25	0.196	0.375	0.375	0.539	0.642	0.646	0.797	0.798	0.997	1.000
			0.30	0.195	0.373	0.374	0.537	0.641	0.645	0.796	0.797	0.997	1.000
			0.35	0.193	0.372	0.373	0.536	0.640	0.644	0.795	0.797	0.997	1.000
		6.0	0.25	0.193	0.372	0.373	0.536	0.641	0.645	0.796	0.797	0.999	1.000
			0.30	0.192	0.371	0.371	0.535	0.640	0.644	0.795	0.796	0.998	1.000
			0.35	0.191	0.370	0.370	0.534	0.639	0.643	0.795	0.795	0.998	1.000
0.4	0.6	4.0	0.25	0.228	0.337	0.507	0.509	0.603	0.739	0.760	0.909	0.978	1.000
			0.30	0.227	0.336	0.506	0.507	0.602	0.738	0.759	0.909	0.977	1.000
			0.35	0.226	0.334	0.505	0.506	0.601	0.737	0.758	0.909	0.977	1.000
		5.0	0.25	0.225	0.333	0.505	0.505	0.600	0.738	0.758	0.910	0.977	1.000
			0.30	0.223	0.331	0.503	0.504	0.599	0.737	0.757	0.910	0.977	1.000
			0.35	0.222	0.330	0.502	0.502	0.597	0.735	0.756	0.909	0.977	1.000
		6.0	0.25	0.222	0.329	0.501	0.502	0.597	0.734	0.756	0.910	0.975	1.000
			0.30	0.220	0.327	0.499	0.501	0.596	0.733	0.755	0.910	0.975	1.000
			0.35	0.219	0.326	0.498	0.500	0.595	0.732	0.754	0.910	0.975	1.000
0.4	0.8	4.0	0.25	0.250	0.317	0.429	0.582	0.588	0.646	0.747	0.772	0.888	1.000
			0.30	0.249	0.316	0.428	0.581	0.587	0.646	0.746	0.771	0.888	1.000
			0.35	0.248	0.314	0.426	0.579	0.586	0.645	0.746	0.770	0.887	1.000
		5.0	0.25	0.246	0.312	0.424	0.577	0.585	0.643	0.743	0.769	0.885	1.000

			0.30	0.245	0.311	0.423	0.576	0.584	0.643	0.742	0.768	0.884	1.000
			0.35	0.244	0.309	0.421	0.574	0.583	0.642	0.742	0.767	0.884	1.000
		6.0	0.25	0.243	0.309	0.420	0.574	0.583	0.641	0.741	0.767	0.883	1.000
			0.30	0.242	0.307	0.419	0.573	0.582	0.640	0.740	0.766	0.883	1.000
			0.35	0.241	0.306	0.417	0.572	0.581	0.640	0.740	0.765	0.883	1.000
0.4	1.0	4.0	0.25	0.315	0.367	0.457	0.581	0.740	0.759	0.806	0.887	0.928	1.000
			0.30	0.314	0.366	0.456	0.580	0.739	0.759	0.805	0.887	0.927	1.000
			0.35	0.313	0.365	0.454	0.578	0.737	0.758	0.805	0.887	0.927	1.000
		5.0	0.25	0.312	0.363	0.452	0.576	0.735	0.759	0.806	0.886	0.926	1.000
			0.30	0.311	0.362	0.450	0.575	0.734	0.758	0.805	0.886	0.926	1.000
			0.35	0.309	0.360	0.449	0.573	0.733	0.758	0.805	0.886	0.925	1.000
		6.0	0.25	0.309	0.360	0.449	0.574	0.734	0.758	0.805	0.886	0.927	1.000
			0.30	0.307	0.359	0.447	0.572	0.732	0.758	0.805	0.886	0.926	1.000
			0.35	0.306	0.357	0.446	0.570	0.731	0.757	0.804	0.886	0.925	1.000
0.6	0.6	4.0	0.25	0.213	0.395	0.395	0.555	0.658	0.660	0.805	0.806	0.998	1.000
			0.30	0.212	0.393	0.394	0.554	0.657	0.659	0.804	0.805	0.998	1.000
			0.35	0.211	0.392	0.392	0.552	0.656	0.657	0.803	0.804	0.999	1.000
		5.0	0.25	0.209	0.389	0.390	0.549	0.653	0.656	0.801	0.802	0.998	1.000
			0.30	0.208	0.387	0.388	0.548	0.652	0.655	0.800	0.801	0.998	1.000
			0.35	0.207	0.386	0.386	0.547	0.651	0.653	0.800	0.801	0.998	1.000
		6.0	0.25	0.206	0.386	0.386	0.546	0.650	0.652	0.799	0.799	0.998	1.000
			0.30	0.205	0.384	0.384	0.544	0.649	0.651	0.798	0.799	0.999	1.000
			0.35	0.203	0.382	0.383	0.543	0.648	0.650	0.797	0.798	0.999	1.000
0.6	0.8	4.0	0.25	0.228	0.357	0.475	0.552	0.584	0.760	0.804	0.822	0.921	1.000
			0.30	0.227	0.356	0.474	0.550	0.583	0.759	0.803	0.822	0.921	1.000
			0.35	0.226	0.355	0.473	0.549	0.582	0.758	0.803	0.821	0.921	1.000
		5.0	0.25	0.224	0.352	0.471	0.547	0.580	0.758	0.803	0.820	0.920	1.000
			0.30	0.223	0.351	0.469	0.546	0.579	0.757	0.802	0.819	0.920	1.000
			0.35	0.222	0.349	0.468	0.544	0.577	0.756	0.801	0.819	0.920	1.000
		6.0	0.25	0.221	0.349	0.468	0.544	0.576	0.756	0.800	0.819	0.920	1.000
			0.30	0.220	0.347	0.466	0.543	0.575	0.755	0.799	0.818	0.919	1.000
			0.35	0.219	0.346	0.465	0.541	0.573	0.754	0.799	0.818	0.919	1.000
0.6	1.0	4.0	0.25	0.256	0.354	0.510	0.568	0.650	0.712	0.786	0.959	0.972	1.000
			0.30	0.256	0.354	0.509	0.567	0.650	0.711	0.785	0.958	0.971	1.000
			0.35	0.255	0.353	0.508	0.566	0.649	0.710	0.784	0.958	0.971	1.000

		5.0	0.25	0.253	0.351	0.506	0.564	0.647	0.709	0.783	0.958	0.971	1.000
			0.30	0.252	0.349	0.504	0.563	0.645	0.707	0.782	0.957	0.970	1.000
			0.35	0.251	0.348	0.502	0.561	0.644	0.706	0.781	0.957	0.970	1.000
		6.0	0.25	0.250	0.347	0.501	0.561	0.644	0.706	0.781	0.957	0.969	1.000
			0.30	0.249	0.345	0.500	0.559	0.642	0.704	0.780	0.957	0.969	1.000
			0.35	0.248	0.344	0.498	0.558	0.641	0.703	0.779	0.956	0.968	1.000
0.8	0.8	4.0	0.25	0.220	0.405	0.405	0.565	0.667	0.669	0.810	0.811	0.998	1.000
			0.30	0.220	0.404	0.405	0.564	0.666	0.668	0.809	0.810	0.998	1.000
			0.35	0.220	0.403	0.404	0.563	0.665	0.667	0.809	0.809	0.998	1.000
		5.0	0.25	0.218	0.400	0.401	0.560	0.663	0.664	0.807	0.808	0.996	1.000
			0.30	0.217	0.399	0.400	0.559	0.662	0.663	0.806	0.807	0.996	1.000
			0.35	0.217	0.398	0.399	0.558	0.661	0.662	0.805	0.806	0.997	1.000
		6.0	0.25	0.215	0.397	0.397	0.556	0.660	0.661	0.805	0.806	0.998	1.000
			0.30	0.214	0.395	0.396	0.555	0.659	0.660	0.804	0.805	0.998	1.000
			0.35	0.213	0.394	0.394	0.553	0.657	0.659	0.803	0.804	0.998	1.000
0.8	1.0	4.0	0.25	0.224	0.364	0.453	0.570	0.570	0.755	0.769	0.829	0.873	1.000
			0.30	0.224	0.364	0.453	0.569	0.570	0.754	0.769	0.829	0.872	1.000
			0.35	0.224	0.363	0.452	0.569	0.569	0.754	0.768	0.828	0.872	1.000
		5.0	0.25	0.222	0.360	0.450	0.567	0.567	0.752	0.767	0.828	0.870	1.000
			0.30	0.222	0.360	0.449	0.566	0.566	0.752	0.766	0.827	0.870	1.000
			0.35	0.221	0.359	0.448	0.564	0.565	0.751	0.765	0.827	0.869	1.000
		6.0	0.25	0.220	0.357	0.447	0.562	0.563	0.750	0.765	0.826	0.870	1.000
			0.30	0.219	0.356	0.445	0.561	0.562	0.749	0.764	0.825	0.869	1.000
			0.35	0.218	0.355	0.444	0.560	0.561	0.747	0.763	0.824	0.869	1.000
1.0	1.0	4.0	0.25	0.223	0.411	0.411	0.572	0.673	0.676	0.815	0.816	0.999	1.000
			0.30	0.224	0.411	0.411	0.572	0.673	0.675	0.815	0.816	0.999	1.000
			0.35	0.224	0.411	0.411	0.572	0.672	0.675	0.815	0.815	0.999	1.000
		5.0	0.25	0.223	0.408	0.409	0.569	0.670	0.673	0.813	0.814	1.000	1.000
			0.30	0.223	0.408	0.408	0.569	0.670	0.673	0.813	0.814	1.000	1.000
			0.35	0.223	0.407	0.408	0.568	0.669	0.672	0.812	0.813	1.000	1.000
		6.0	0.25	0.221	0.406	0.406	0.566	0.667	0.670	0.811	0.812	1.000	1.000
			0.30	0.221	0.405	0.405	0.565	0.666	0.669	0.810	0.811	1.000	1.000
			0.35	0.220	0.404	0.404	0.564	0.665	0.668	0.810	0.810	1.000	1.000



Rysunek 3.37 Znormalizowane wartości pierwszych dziesięciu częstotliwości własnych względem dziesiątej częstotliwości własnej dla wszystkich analizowanych wymiarów płyt

W oparciu o powyższy wykres oraz przedstawione w tym punkcie wartości (Tablica 3.1) można stwierdzić, że dla ustalonych wymiarów szyby próżniowej, znormalizowana wartość częstotliwości drgań własnych niemalże nie ulega zmianie, pomimo różnych wartości grubości tafli szkła oraz wysokości próżni. Znacznie większy wpływ na różnicę pomiędzy kolejnymi wartościami częstotliwości drgań własnych mają wymiary całkowite płyty typu VIG. Pomimo tego, że wpływ ten jest większy, w oparciu o powyższy rysunek (Rysunek 3.37) można stwierdzić, że względna zmiana kolejnych znormalizowanych wartości częstotliwości drgań własnych w odniesieniu do dziesiątej częstotliwości drgań własnych nie jest znacząca. W obrębie danej postaci drgań własnych, względne różnice znormalizowanych wartości częstotliwości drgań własnych oscylują w granicach do około 8%.

4. Metody sztucznej inteligencji

4.1. Wprowadzenie

Jako sztuczną inteligencję (AI; ang. Artificial Intelligence) rozumie się inteligencję, którą wykazać może sztucznie stworzone urządzenie. Twórcą tego terminu jest John McCarthy, który przedstawił go na konferencji w Dartmouth w 1956 roku. Andreas Kaplan i Michael Haenlein zdefiniowali ten termin jako „zdolność systemu do prawidłowego interpretowania danych pochodzących z zewnętrznych źródeł, nauki na ich podstawie oraz wykorzystywania tej wiedzy, aby wykonywać określone zadania i osiągać cele poprzez elastyczne dostosowanie"[23].

Poza opisanymi w niniejszej pracy metodami AI istnieje wiele innych takich jak przetwarzanie języka naturalnego, widzenie komputerowe, reprezentacja wiedzy i wnioskowanie, robotyka, planowanie oraz inteligencja społeczna. Niemniej jednak, w tym rozdziale zostały opisane dwa algorytmy, które wykorzystano do stworzenia modeli predykcyjnych (modeli, których celem jest przewidzenie danej wartości na podstawie dostarczonych zmiennych). Pierwszy z nich opiera się na metodzie Extreme Gradient Boosting (XGB, XGBoost; z ang. maksymalne wzmacnianie gradientu) zaliczanej do zbioru algorytmów uczenia maszynowego (ML; ang. Machine Learning). Drugi z nich opiera się na głębokich sieciach neuronowych (DNN; ang. Deep Neural Networks) zaliczanych do zbioru algorytmów uczenia głębokiego (DL; ang. Deep Learning). Zbiór metod DL jest podzbiorem zbioru metod ML, natomiast zbiór metod ML jest podzbiorem zbioru metod AI (Rysunek 4.1).

Rysunek 4.1 Schemat klasyfikacyjny opisywanych metod sztucznej inteligencji

Wszystkie tworzone modele AI, w celu osiągnięcia pożądanej przez użytkownika funkcjonalności, wymagają zbiorów danych. Dzięki nim, modele mogą

przeprowadzić proces uczenia. Im większy zbiór danych, tym większa szansa na powodzenie działania stworzonego algorytmu.

Mając na uwadze fakt, że algorytmy uczenia głębokiego są w zasadzie również algorytmami uczenia maszynowego, oczywistym jest, że mają one ogromną ilość podobieństw. Aczkolwiek, można wyróżnić również pewne aspekty, które je rozróżniają. Stworzenie modelu przy użyciu algorytmów ML jest zdecydowanie mniej złożone, niż w przypadku algorytmów DL. Modele te wymagają przeważnie mniejszej ilości danych do procesu trenowania, ale o odpowiedniej strukturze (zwykle wartości liczbowe) i są zdecydowanie mniej wymagające pod względem sprzętowym. Proces trenowania zajmuje zazwyczaj niewiele czasu. Potrzebują one natomiast większej ingerencji człowieka w celu osiągnięcia pożądanych rezultatów i są przeważnie wykorzystywane do mniej skomplikowanych zagadnień. Modele stworzone za pomocą algorytmów DL wymagają dużo większej ilości danych, które nie muszą posiadać odpowiedniej struktury (poza wartościami liczbowymi mogą to być również tekst lub dźwięk), ale wymagania sprzętowe są zdecydowanie większe. Proces trenowania modelu zajmuje przeważnie bardzo dużo czasu. Tego typu modele wykorzystuje się do bardzo złożonych zagadnień.

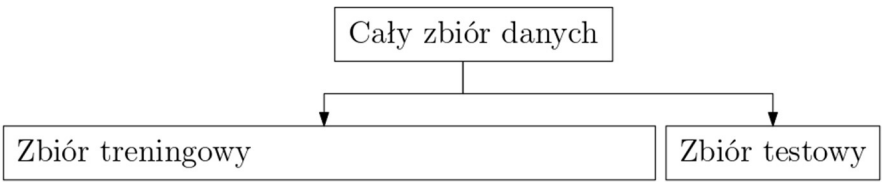
W niniejszym rozdziale w pierwszym kroku omówiono przyjęte zbiory danych. Dopiero w kolejnych krokach opisano wykorzystane metod AI (XGB oraz DNN) w oparciu o te dane.

4.2. Zbiory danych

4.2.1. Podział zbioru danych

Jednym z najbardziej kluczowych aspektów tworzenia efektywnego modelu predykcyjnego jest odpowiedni dobór zbioru danych, dzięki którym model przeprowadzi proces uczenia się. Jest to szczególnie ważne, ponieważ celem algorytmu jest znalezienie przewidywalnego wzoru ukrytego w całym zbiorze danych.

Cały zbiór danych można podzielić na następujące dwa podzbiory – treningowy oraz testowy (Rysunek 4.2).



Rysunek 4.2 Schemat podziału całego zbioru danych

Zbiór danych treningowych to zestaw danych, które podlegają analizie. Model uczy się na ich podstawie, w celu zaprojektowania odpowiednich reguł. Jest on wykorzystywany również do oceny efektywności reguł modelu oraz służy dostrojeniu parametrów i funkcji wejściowych modelu predykcyjnego.

Zbiór danych testowych to zestaw danych, które nie zostały użyte do trenowania modelu. Przy jego użyciu przeprowadza się ostateczny test, gdy ostateczny model zostanie już ustalony, aby uzyskać możliwie najlepsze oszacowanie skuteczności stworzonego modelu w oparciu o zupełnie nowe dane.

Celem dzielenia danych jest zapobiegnięcie:

- niepożądanej ekstrapolacji (budowanie modelu na podstawie danych, które nie powinny być znane; zapobieganie otrzymywaniu wyników poza granicami stosowalności modelu);
- nadmiernemu dopasowaniu (ang. overfitting; proces projektowania modelu, który dostosowuje się do dostarczonego zbioru danych tak ściśle, że staje się nieefektywny w przyszłości);
- niedopasowaniu (ang. underfitting; proces projektowania modelu, który dostosowuje się do dostarczonego zbioru danych tak luźno, że staje się nieefektywny w przyszłości).

4.2.2. Przyjęty zbiór danych

Przyjęte zbiory danych składają się z elementów mających sens fizyczny. Są to zarówno wartości skalarne z jednostkami jak i bez jednostek. Dane, na podstawie których modele oparte na algorytmach AI mogą zostać wytrenowane muszą być wartościami bezwymiarowymi. Tablica 4.1 zawiera zestawione i opisane wszystkie wartości danych wejściowych oraz wyjściowych przyjętych w tej pracy.

Tablica 4.1 Opis wartości z przyjętych zbiorów danych

Oznaczenie parametru	Jednostka	Opis
DANE WEJŚCIOWE		
A	m	wymiar płyty w kierunku X
B	m	wymiar płyty w kierunku Y
t_g	mm	grubość tafli szkła
t_v	mm	grubość warstwy próżni
w_s	mm	szerokość uszczelki
n_x	-	ilość pilarków w kierunku X
n_y	-	ilość pilarków w kierunku Y
d_p	mm	średnica pilarka

x_p	*mm*	współrzędna X pierwszego pilarka
y_p	*mm*	współrzędna Y pierwszego pilarka
ρ_g	*kg/m³*	gęstość szkła
E_g	*GPa*	moduł sprężystości podłużnej szkła
ν_g	-	współczynnik Poissona szkła
ρ_p	*kg/m³*	gęstość stali (materiał pilarków)
ν_p	-	współczynnik Poissona stali (materiał pilarków)
ρ_s	*kg/m³*	gęstość stali (materiał uszczelki)
E_s	*GPa*	moduł sprężystości podłużnej stali (materiał uszczelki)
ν_s	-	współczynnik Poissona stali (materiał uszczelki)
$f_1 \div f_i$	*Hz*	wartość częstotliwości dla kolejnych postaci drgań własnych od 1 do i
DANE WYJŚCIOWE		
E_p	*GPa*	moduł sprężystości podłużnej stali (materiał pilarków)

Celem stworzenia zbioru danych do trenowania modeli predykcyjnych opisanych w tym rozdziale, określono które z powyżej wymienionych zmiennych (Tablica 4.1) zostaną wykorzystane. Tablica 4.2 przedstawia finalnie przyjęte w tym celu zmienne.

Tablica 4.2 Zestawienie finalnie przyjętych zmiennych w zbiorze danych

Oznaczenie parametru	Jednostka	Oznaczenie w zbiorze danych
DANE WEJŚCIOWE		
A	*m*	x_1
B	*m*	x_2
t_g	*mm*	x_3
t_v	*mm*	x_4
n_x	-	x_5
n_y	-	x_6
x_p	*mm*	x_7
y_p	*mm*	x_8
$f_1 \div f_{30}$	*Hz*	$x_9 \div x_{38}$
DANE WYJŚCIOWE		
E_p	*GPa*	y

Ponadto, z uwagi na specyfikę sieci neuronowych wykorzystywanych do stworzenia modelu predykcyjnego, dla przyjętego zbioru danych należy dokonać normalizacji.

Wartości własne (częstotliwości drgań własnych) dla danego zbioru parametrów uzyskano za pomocą modeli numerycznych opisanych w rozdziale 3. Tablica 4.3 przedstawia wszystkie wartości przyjęte do wykonania modeli numerycznych, na podstawie których wyznaczono wartości częstotliwości drgań własnych. Sumarycznie stworzono 360 różnych modeli numerycznych, poprzez wykonanie kombinacji uwzględniających wszystkie podane wartości.

Tablica 4.3 Zestawienie wartości wykorzystanych do stworzenia modeli numerycznych

Oznaczenie parametru	Jednostka	Przyjęte wartości	Ilość kombinacji
A	m	0.40; 0.60; 0.80; 1.00	10
B	m	0.40; 0.60; 0.80; 1.00	
t_g	mm	4.0; 5.0; 6.0	3
t_v	mm	0.25; 0.30; 0.35	3
w_s	mm	9.0	1
n_x	-	Wartości uzależnione od wymiaru płyty: 0.40 – 6; 0.60 – 10; 0.80 – 14; 1.00 – 17	-
n_y	-	Wartości uzależnione od wymiaru płyty: 0.40 – 6; 0.60 – 10; 0.80 – 14; 1.00 – 17	
d_p	mm	0.6	1
x_p	mm	Wartości uzależnione od wymiaru płyty: 0.40 – 62.5; 0.60 – 52.5; 0.80 – 42.5; 1.00 – 60.0	-
y_p	mm	Wartości uzależnione od wymiaru płyty: 0.40 – 62.5; 0.60 – 52.5; 0.80 – 42.5; 1.00 – 60.0	
ρ_g	kg/m^3	2500.0	1
E_g	GPa	72.0	1
ν_g	-	0.22	1
ρ_p	kg/m^3	7850.0	1
ν_p	-	0.31	1
ρ_s	kg/m^3	7850.0	1
E_s	GPa	210.0	1
ν_s	-	0.31	1
E_p	GPa	190.0; 200.0; 210.0; 220.0	4
		Sumaryczna ilość wszystkich kombinacji	**360**

Cały zbiór danych został losowo podzielony na część treningową (80%) oraz testową (20%).

4.3. XGBoost

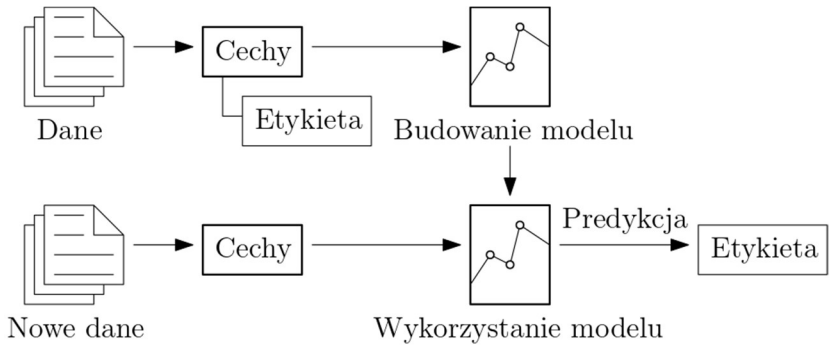
4.3.1. Opis metody

Metoda Extreme Gradient Boosting jest jednym z wielu algorytmów uczenia maszynowego (ML). Jest to metoda wzmacniania gradientu oparta na drzewach decyzyjnych (ang. Gradient Boosting Decision Trees – GBDT). Zapewnia równoległe wzmacnianie drzewa i jest wiodącą biblioteką uczenia maszynowego dla problemów związanych z regresją, klasyfikacją, predykcją oraz rankingiem.

W celu zrozumienia zasady działania XGB należy zrozumieć następujące koncepcje i algorytmy ML, na których oparta jest opisywana metoda.

A. Nadzorowane uczenie maszynowe

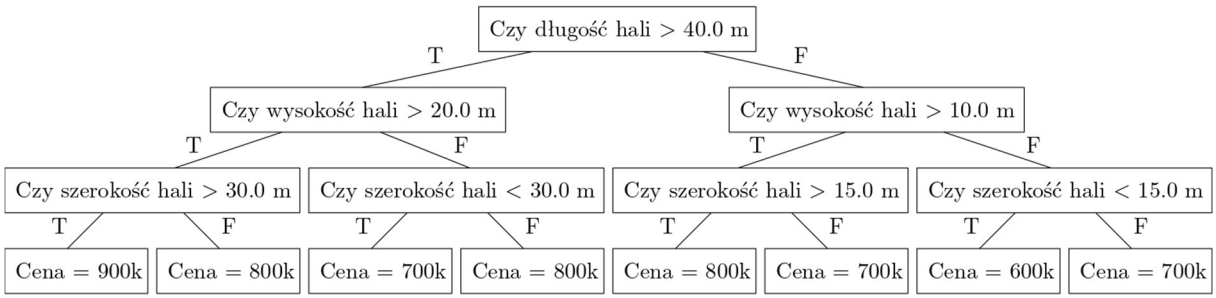
Nadzorowane uczenie maszynowe polega na wykorzystaniu algorytmów w celu wytrenowania modelu do znajdowania wzorców w zbiorach danych z etykietami i cechami, a w kolejnym etapie polega na wykorzystaniu wytrenowanego modelu do przewidywania etykiet dla atrybutów nowego zestawu danych (Rysunek 4.3).



Rysunek 4.3 Schemat działania nadzorowanego uczenia maszynowego

B. Drzewa decyzyjne

Modele składają się z drzew decyzyjnych, które przewidują etykietę poprzez przejście procedury drzewa (odpowiadając na pośrednie pytania) i oszacowując minimalną liczbę pytań wymaganą do oceny prawdopodobieństwa podjęcia prawidłowej decyzji. Drzewa te mogą zostać użyte zarówno w celu zakwalifikowania elementu do kategorii, jak i w celu przewidzenia ciągłej wartości liczbowej. Poniżej przedstawiono przykład ilustrujący przykładowe drzewo decyzyjne (Rysunek 4.4).



Rysunek 4.4 Przykładowe drzewo decyzyjne

C. Uczenie zespołowe

GBDT to algorytm uczenia zbioru drzew decyzyjnych, podobny do losowego lasu (ang. Random Forest – RF), do klasyfikacji i regresji. Celem algorytmów uczenia zespołowego jest połączenie wielu algorytmów uczenia maszynowego, aby uzyskać możliwie jak najlepszy model.

RF, podobnie jak GDBT, polega na zbudowaniu modelu składającego się z wielu drzew decyzyjnych. Różnicą jest jednak sposób budowy i łączenia drzew. RF wykorzystuje technikę o nazwie *bagging*.

Zarówno RF, jak i GBDT budują model składający się z wielu drzew decyzyjnych. Różnica polega na sposobie budowy i łączenia drzew. Las losowy wykorzystuje technikę o nazwie *bagging* (agregacja bootstrapowa) do równoległego budowania pełnych drzew decyzyjnych z losowych próbek zbioru danych. Ostateczna predykcja jest średnią wszystkich predykcji drzewa decyzyjnego.

D. Wzmacnianie gradientu.

Pojęcie wzmacniania gradientu (ang. Gradient Boosting – GB) wywodzi się ze wzmacniania słabego modelu poprzez połączenie go z wieloma innymi słabymi modelami. Celem jest wspólne wygenerowanie silnego modelu. GB jest to rozszerzenie wzmacniania (ang. boosting), w którym proces addytywnego generowania słabych modeli jest sformalizowany przy użyciu metody gradientu prostego (ang. gradient descent) na funkcji celu. GB wyznacza docelowe wyniki dla następnego modelu, dzięki którym osiągnięty błąd będzie jak najmniejszy. Wyniki te są otrzymywane na podstawie wartości gradientu wspomnianego błędu, odniesionego do predykcji.

GBDT w sposób iteracyjny trenują zestaw nieznacznie rozbudowanych drzew decyzyjnych, przy czym w każdej iteracji wykorzystywane są resztkowe błędy z modelu poprzedniego celem dopasowania do modelu następnego. Ostateczną predykcją jest ważona suma wszystkich predykcji drzewa decyzyjnego. W przypadku RF za zminimalizowanie wariancji oraz nadmiernego dopasowania odpowiedzialna jest agregacja bootstrapowa, natomiast w przypadku GBDT to proces wzmacniania minimalizuje skośność i niedopasowanie.

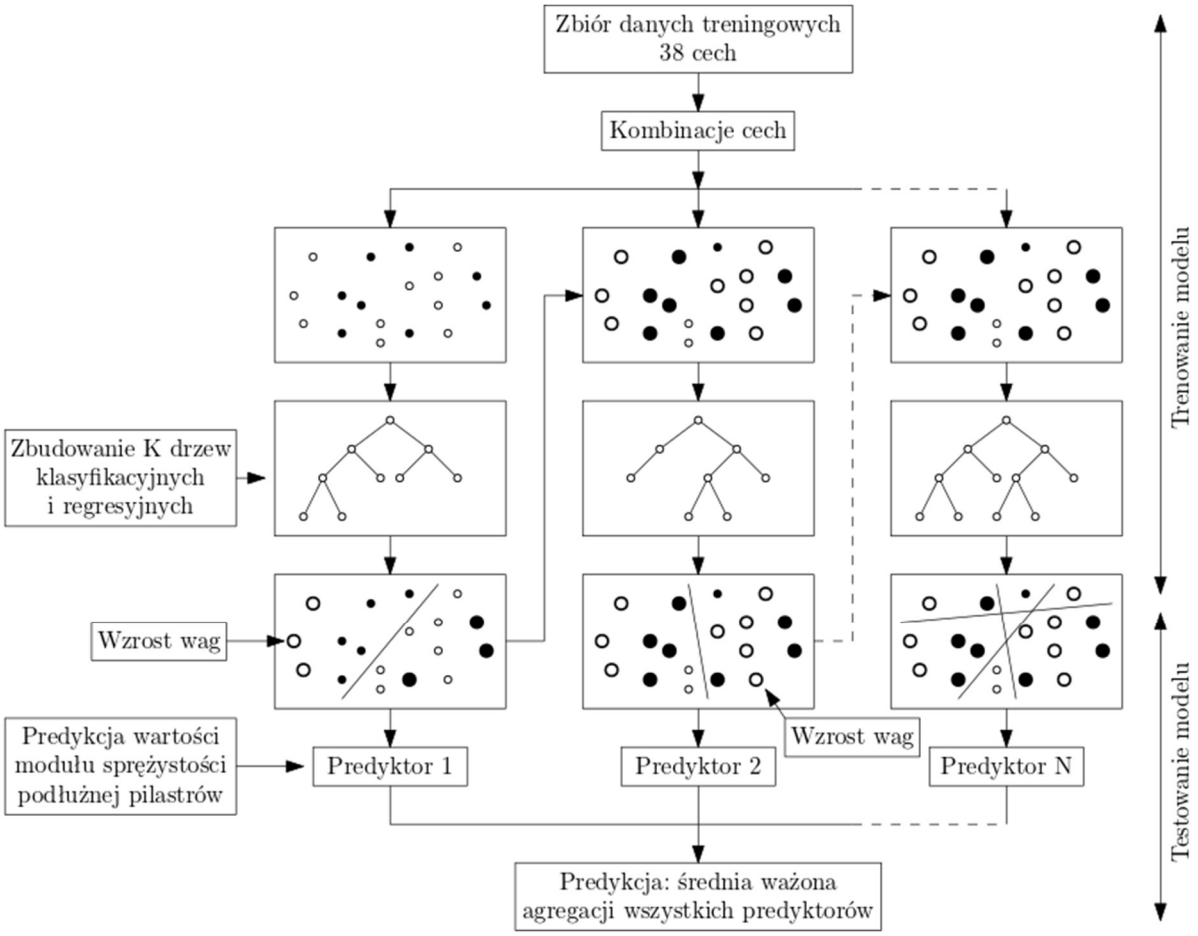
Metoda XGB jest to skalowalna i bardzo dokładna implementacja zwiększenia gradientu, rozciągająca granice mocy obliczeniowej algorytmów wzmacnianych drzew decyzyjnych. Została ona stworzona przede wszystkim w celu zwiększenie wydajności i szybkości obliczeniowej modeli uczenia maszynowego. Odmiennie niż w przypadku GBDT, w XGBoost drzewa decyzyjne są budowane równolegle, a nie sekwencyjnie. W metodzie tej wartości gradientów są skanowane, a następnie te sumy częściowe wykorzystuje się do ocen jakości podziałów w każdym możliwym podziale zbioru wykorzystywanego do trenowania modelu.

Metoda XGB jako rozwinięcie metody Gradient Boosting (GB; z ang. wzmacnianie gradientu), była początkowo projektem badawczym Tianqi Chen [24] w ramach grupy badawczej o nazwie Distributed (Deep) Machine Learning Community (DMLC). Zyskała ona znacząco na popularności po jej użyciu w ramach zwycięskiego rozwiązania w konkursie Higgs Machine Learning Challenge. Niedługo później metoda została zaimplementowana jako pakiety dla wielu języków programowania, a następnie zintegrowana z wieloma innymi pakietami, co zdecydowanie ułatwiło korzystanie z niej.
Należy również wspomnieć, że w 2019 roku metoda XGB była jednym ze zdobywców prestiżowej nagrody Technologii Roku InfoWorld.

Stosowanie tej metody niesie ze sobą bardzo wiele korzyści. Biblioteka XGB jest stale rozwijana na całym świecie z uwagi na nieustannie wzrastającą liczbę naukowców, którzy się do tego przyczyniają. Modele uczenia maszynowego XGBoost świetnie łączą ze sobą wydajność oraz czas przetwarzania, w odniesieniu do innych algorytmów, dlatego są one często używane przez naukowców zajmujących się różnego rodzaju danymi. Ponadto, można jej używać na platformach takich jak Windows, Linux czy OS X.

4.3.2. Opis modelu

Uproszczona struktura stworzonego algorytmu XGB została przedstawiona na poniższym rysunku (Rysunek 4.5), a jej dokładny opis znajduje się w dalszej części tego podpunktu [25].



Rysunek 4.5 Uproszczona struktura stworzonego algorytmu

Założono, że zbiorem danych jest:

$$D = \{(x_i, y_i)\};\ i = 1,2, \dots, n. \tag{4.1}$$

Model, który zostanie wykorzystany do trenowania i uczenia składa się z K drzew.

Wynik otrzymany za pomocą tego modelu można opisać następująco:

$$\hat{y} = \emptyset(x_i) = \sum_{k=1}^{K} f_k(x_i)\,;\; f_k \in F, \tag{4.2}$$

gdzie F jest hipotetyczną przestrzenią, a $f(x)$ jest regresyjnym drzewem decyzyjnym:

$$\mathrm{F} = \{f(x) = \omega_{q(x)}\}. \tag{4.3}$$

W równaniu (4.3), $q(x)$ jest liściem, czyli węzłem końcowym drzewa decyzyjnego, x-tej próbki, a ω jest wynikiem tego liścia. Przewidywany wynik dla t-tej iteracji można określić następująco:

$$\hat{y}_i^t = \hat{y}_i^{t-1} + f_t(x_i). \tag{4.4}$$

Funkcję celu można zatem określić jako:

$$J(f_t) = \sum_{i=1}^{n} L\left(y_i, \hat{y}_i^{t-1} + f_t(x_i)\right) + \Omega(f_t), \tag{4.5}$$

gdzie L jest funkcją błędu, a $\Omega(f_t)$ odwzorowuje złożoność modelu.
T określa całkowitą ilość liści, a ω wynik liścia:

$$\Omega(f_t) = \gamma T_t + \lambda \frac{1}{2} \sum_{j=1}^{T} \omega_j^2. \tag{4.6}$$

Następnie, równanie (4.5) można uprościć przez zastosowanie rozwinięcia Taylora drugiego rzędu:

$$J(f_t) = \sum_{i=1}^{n} \left[L(y_i, \hat{y}_i^{t-1}) + g_i f_i(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t), \tag{4.7}$$

$$g_i = \frac{\partial L(y_i, \hat{y}_i^{t-1})}{\partial \hat{y}_i^{t-1}}, \tag{4.8}$$

$$h_i = \frac{\partial^2 L(y_i, \hat{y}_i^{t-1})}{\partial \hat{y}_i^{t-1^2}}. \tag{4.9}$$

W oparciu o powyższą analizę, finalną funkcję celu można zapisać jako:

$$J(f_t) = \sum_{i=1}^{n} \left[g_i \omega_{q(x_i)} + \frac{1}{2} h_i \omega_{q(x_i)}^2 \right] + \gamma T_t + \lambda \frac{1}{2} \sum_{j=1}^{T} \omega_j^2. \tag{4.10}$$

Podsumowując, funkcja celu jest zoptymalizowana, a optymalne rozwiązanie jest następujące:

$$\omega_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda}, \tag{4.11}$$

$$J(f_t) = -\frac{1}{2}\sum_{j=1}^{T}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T. \tag{4.12}$$

W celu określenia efektywności modelu wyznaczono wartość pierwiastka błędu średniokwadratowego (ang. Root Mean Square Deviation – RMSE). Równanie określające opisane wyrażenie jest następujące:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \bar{y})^2}, \tag{4.13}$$

gdzie $y^{(i)}$ oznacza docelową wartość wykorzystywaną do trenowania modelu, natomiast $\bar{y}$ oznacza wartość przewidywaną przez model.

Do stworzenia algorytmu XGB w języku Python wykorzystano następujące biblioteki: *xgboost* [26], *sklearn* [27], *openpyxl* [28], *pandas* [29], *numpy* [30], *itertools* [31], *tqdm* [32] oraz *matplotlib* [33]. Dla stworzonego modelu przeprowadzono proces uczenia przy użyciu ustalonego wcześniej zbioru danych (punkt 4.2.2). Tablica 4.4 przedstawia przyjęte w modelu parametry wraz z ich opisem.

Tablica 4.4 Opis parametrów modelu predykcyjnego XGB

Parametr	**Opis parametru**
Ilość estymatorów (drzew decyzyjnych) (ang. number of estimators)	Ilość drzew decyzyjnych stworzona w obrębie modelu predykcyjnego.
Współczynnik gamma (ang. gamma ratio)	Minimalna redukcja strat wymagana do wykonania kolejnego podziału na węźle liścia drzewa. Im większa będzie wartość tego współczynnika, tym bardziej konserwatywny będzie algorytm.
Współczynnik uczenia (ang. learning rate)	Wartość zmniejszenia rozmiaru skoku, używana przy aktualizacji, aby zapobiec nadmiernemu dopasowaniu.
Maksymalna głębokość pojedynczego drzewa (ang. max depth)	Maksymalna ilość poziomów pojedynczej gałęzi drzewa decyzyjnego.
Polityka wzrostu (ang. grow policy)	Kontroluje sposób dodawania nowych węzłów do drzewa decyzyjnego. Istnieją dwie możliwe opcje: depthwise – podział w węźle będącym najbliżej głównej gałęzi; lossguide – podział w węźle z największą zmianą wartości błędu.

Współczynnik podziału kolumny w drzewie (ang. colsample by tree)	Stosunek podpróbkowania kolumn podczas konstruowania każdego drzewa. Podpróbkowanie odbywa się raz dla każdego skonstruowanego drzewa.
Współczynnik podziału podpróbek (ang. subsample)	Stosunek podpróbek elementów zbioru uczącego. Częściowy wybór próbek z całego zbioru uczącego, w celu zapobiegnięcia nadmiernemu dopasowaniu.

Celem znalezienia najbardziej optymalnych parametrów modelu predykcyjnego, przeanalizowano wiele kombinacji różnych wartości tych parametrów. Parametry końcowe wybrano celem uzyskania najmniejszego możliwego błędu średniokwadratowego (RMSE), który został określony równaniem (4.13). Tablica 4.5 przedstawia analizowane oraz finalnie przyjęte wartości omawianych parametrów.

Tablica 4.5 Przyjęte parametry modelu predykcyjnego XGB

Parametr	**Zakres wartości**	**Przyjęta wartość**
Ilość estymatorów (drzew decyzyjnych)	50, 100, 200, 500, 1000, 1500, 2000, 2500	500
Współczynnik gamma	0.0, 0.1, 0.2	0.0
Współczynnik uczenia	0.01, 0.1, 0.2, 0.23, 0.26, 0.29, 0.32, 0.35	0.2
Maksymalna głębokość pojedynczego drzewa	3, 5, 10, 20, 30	10
Polityka wzrostu	Depthwise, Lossguide	Depthwise
Współczynnik podziału kolumny w drzewie	0.3, 0.6, 1.0	1.0
Współczynnik podziału podpróbek	0.3, 0.6, 1.0	1.0

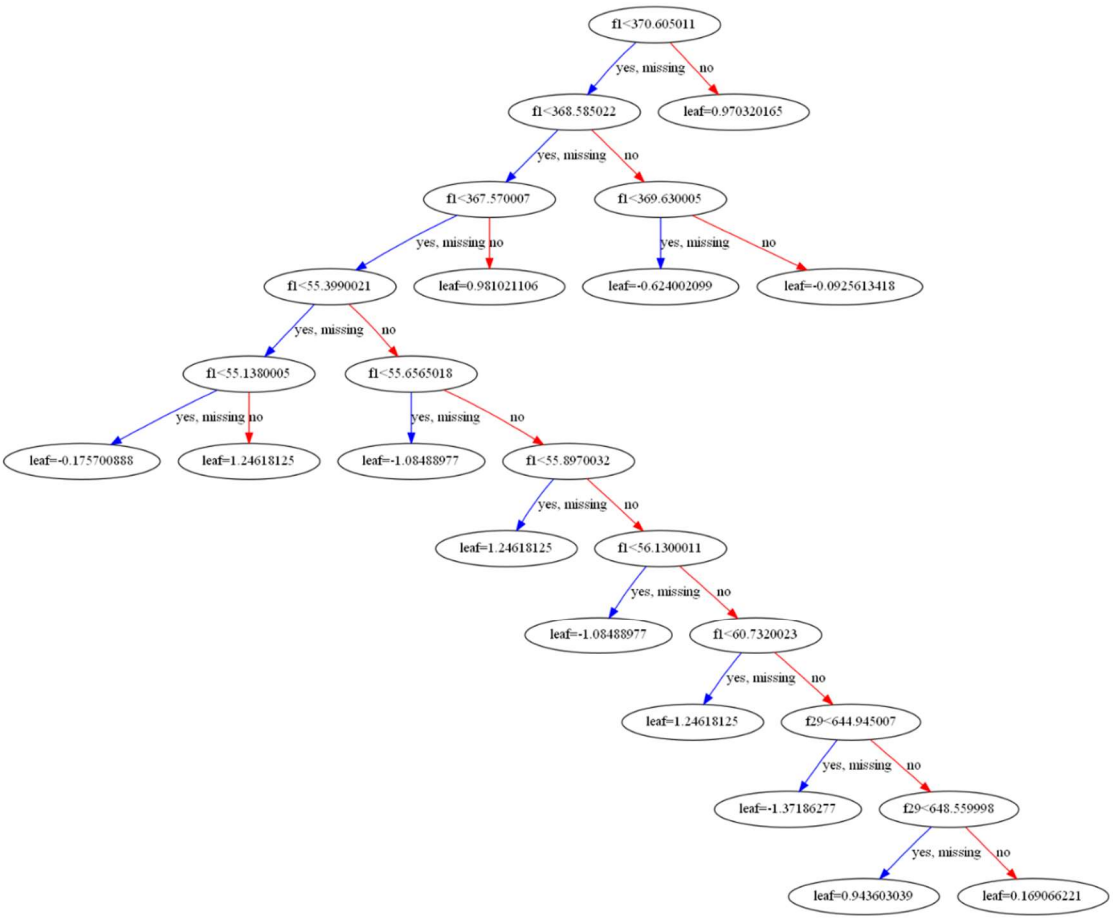
4.3.3. Wyniki

Błędy otrzymane dla przyjętych wartości parametrów modelu predykcyjnego metody XGB zaprezentowano poniżej (Tablica 4.6). Wyznaczono je zarówno dla zbioru danych treningowych jak i testowych.

Tablica 4.6 Błędy otrzymane dla modelu predykcyjnego XGB

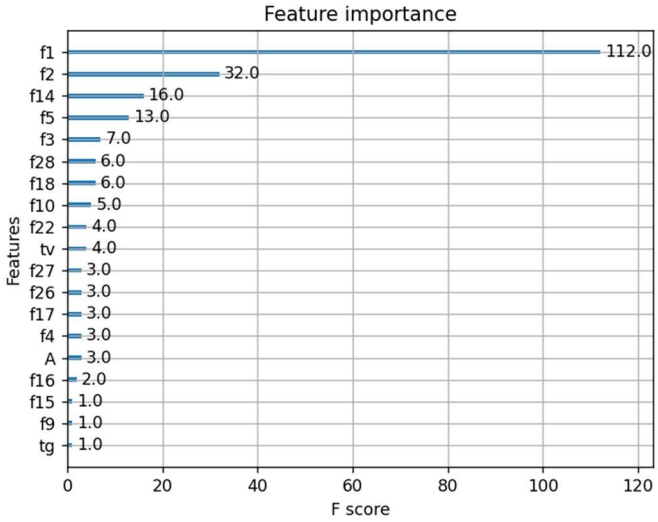
Zbiór danych	**RMSE [GPa]**
Dane treningowe	0.0025
Dane testowe	12.0836

Rysunek 4.6 przedstawia przykładowe drzewo decyzyjne ze zbioru wszystkich stworzonych automatycznie drzew decyzyjnych.



Rysunek 4.6 Przykładowe drzewo decyzyjne stworzonego modelu XGB

Rysunek 5.9 przedstawia wykres ważności cech (przyjętych zmiennych wejściowych) dla stworzonego modelu predykcyjnego metody XGB.



Rysunek 4.7 Wykres ważności cech dla stworzonego modelu XGB

W załączniku nr 2 znajduje się kod źródłowy modelu stworzonego przy użyciu algorytmu XGBoost.

4.4. Głębokie sieci neuronowe

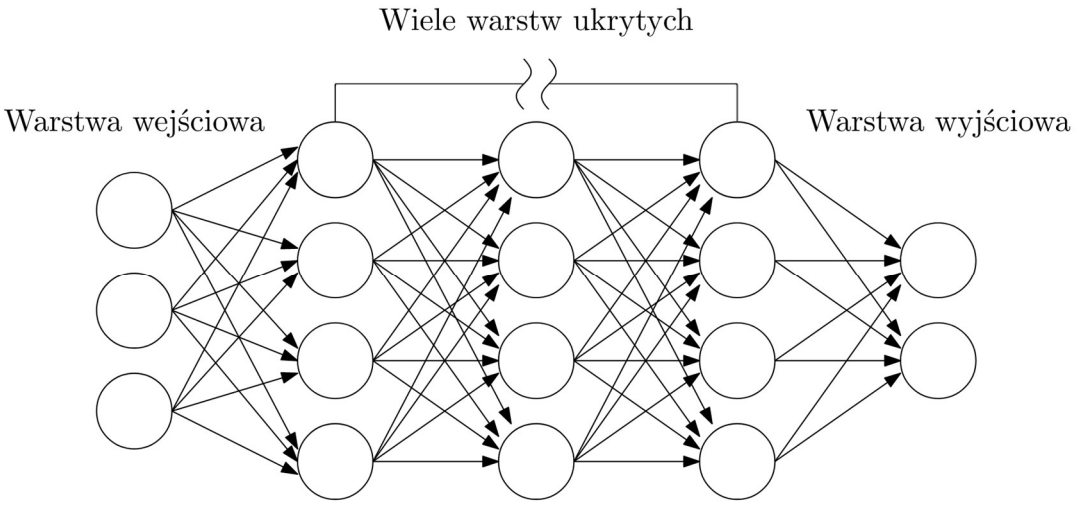
4.4.1. Opis metody

Sieci neuronowe (ang. Neural Networks – NN) są częścią zagadnienia jakim jest ML, ponadto są podstawą algorytmów DL. Niekiedy używa się również określeń takich jak sztuczna sieć neuronowa (ang. Artificial Neural Network – ANN) czy symulowana sieć neuronowa (ang. Simulated Neural Network – SNN). Ich celem jest komputerowe odwzorowanie struktury sieci biologicznych neuronów znajdujących się w mózgu człowieka, które się wzajemnie komunikują, aby umożliwić urządzeniom rozpoznawanie wzorców w zbiorach danych.

Składają się one z następujących warstw węzłów (neuronów):

- warstwa wejściowa (ang. input layer),
- jedna bądź więcej warstw ukrytych (ang. hidden layers),
- warstwa wyjściowa (ang. output layer).

Każdy sztuczny neuron danej warstwy łączy się z każdym sztucznym neuronem sąsiednich warstw (lub warstwy) oraz posiada z nim powiązaną wagę i próg aktywacji. Jeżeli sieć neuronowa składa się z minimum czterech warstw – warstwy wejściowej, warstwy wyjściowej oraz co najmniej dwóch warstw ukrytych – wtedy można określić ją mianem głębokiej sieci neuronowej (ang. Deep Neural Network – DNN) (Rysunek 4.8).



Rysunek 4.8 Schemat głębokiej sieci neuronowej

Pomimo swojej popularności, sieci neuronowe nie są zagadnieniem bardzo młodym, a ich historia, wbrew pozorom, nie jest krótka. Pierwszym krokiem w kierunku stworzenia czegoś na kształt sieci neuronowych w ludzkim mózgu było badanie opublikowane w 1943 roku [34]. McCulloch i Pitts porównali w nim neurony z progiem binarnym do logiki boolowskiej.

W 1958 roku Frank Rosenblatt wykonał kolejny krok do przodu, względem pracy McCulloch i Pitts [34], poprzez dodanie do neuronu równania uwzględniającego wagę. Stworzył on i opisał pojęcie perceptronu, a następnie wykorzystał je do stworzenia algorytmu, którego celem było rozróżnienie kart zaznaczonych po lewej oraz prawej stronie [35].

Kolejnym istotnym aspektem było wykorzystanie propagacji wstecznej. Mimo, że była ona już wcześniej znana, po raz pierwszy w sieciach neuronowych dokonał tego Paul Werbos w 1974 roku [36].

Kolejnym ważnym krokiem była praca opublikowana w 1989 roku, której autorem był Yann LeCun [37]. Przedstawił on w niej sposób umożliwiający trenowanie algorytmów poprzez wykorzystanie ograniczeń w propagacji wstecznej i połączeniu jej z architekturą sieci neuronowych. Stworzony przez autora algorytm został użyty celem rozpoznania przez komputer ręcznie zapisanych na przesyłkach cyfr kodu pocztowego, które były dostarczane przez Urząd Pocztowy Stanów Zjednoczonych.

Z czasem, powstających prac, których tematyka dotyczyła sieci neuronowych, było coraz więcej. Ich zastosowanie stale rosło i nadal rośnie. Pomimo dużego wykorzystania sieci neuronowych w wielu aspektach życia codziennego, tematyka ta jest nadal rozwijana i udoskonalana, a jej celem jest osiągnięcie efektywności ludzkiego mózgu.

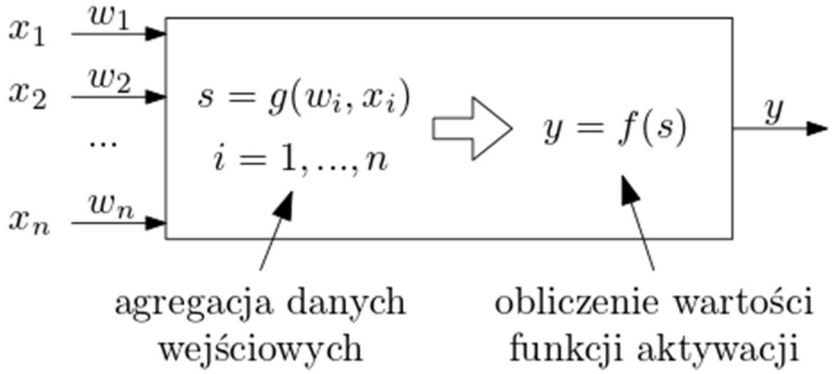
Algorytmy NN są również wykorzystywane w inżynierii, między innymi w celu uzyskania właściwości geotechnicznych gruntu lub właściwości mechanicznych systemów warstwowych na podstawie nieniszczących badań eksperymentalnych [38].

W celu poprawnego zrozumienia działania sieci neuronowych należy zrozumieć jej podstawowe pojęcia. W 2015 roku Szaleniec i Tadeusiewicz opracowali zbiór pojęć związanych z sieciami neuronowymi [39]. Poniżej zestawiono i opisano najważniejsze z nich.

A. Neuron

Neuron jest podstawowym elementem składowym sieci neuronowej. Jego zadaniem jest przetworzyć otrzymane informacje. Jest on bardzo dużym uproszczeniem biologicznej komórki nerwowej.

Każdy neuron w swej strukturze posiada dowolną ilość wejść oraz jedno wyjście. Jego istotną częścią jest zbiór wag, których wartości ustalane są w trakcie procesu uczenia (Rysunek 4.9).

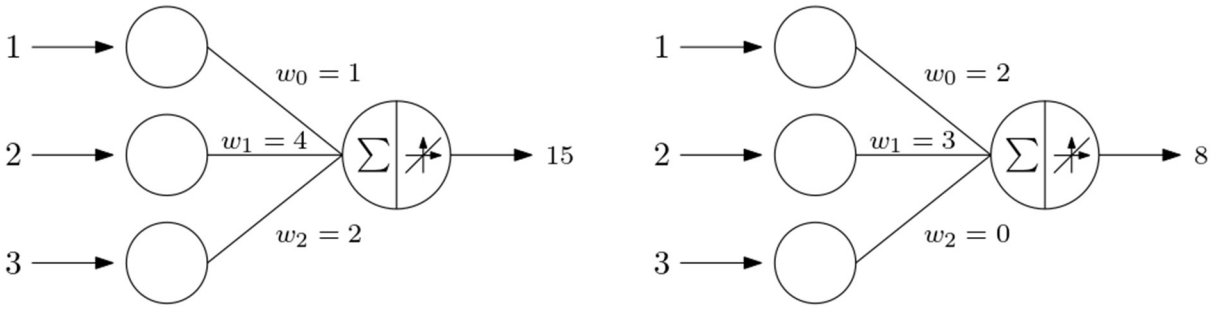


Rysunek 4.9 Schemat neuronu

W neuronie wykonywana jest agregacja danych wejściowych, przy uwzględnieniu wag oraz stworzenie sygnału wyjściowego. Biorąc pod uwagę różne sposoby agregacji oraz formy funkcji aktywacji, wyróżnić można różne rodzaje neuronów. Jako najczęściej wykorzystywane uznać można neurony liniowe, neurony radialne, neurony sigmoidalne oraz odmianę neuronów sigmoidalnych – neurony tangensoidalne.

B. Wagi

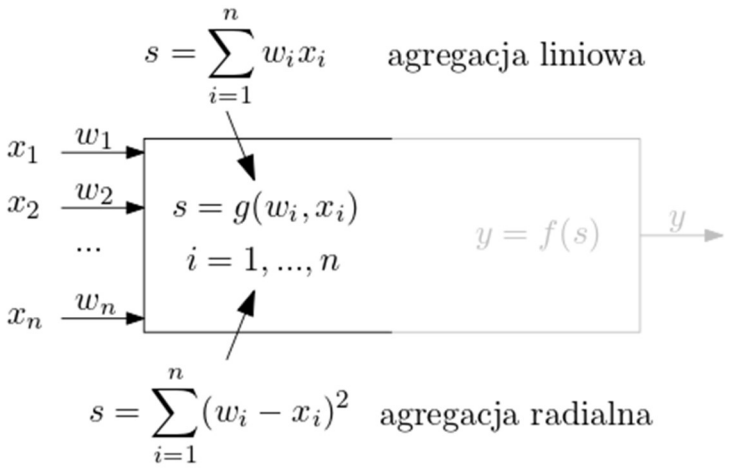
Wagami określa się parametry neuronu, które decydują o jego właściwościach oraz roli w procesie rozwiązywania przez sieć postawionego zadania. Zazwyczaj, przyjęty algorytm uczenia lub samouczenia ustala wagi całej sieci neuronowej. W zależności od wartości wejściowych danych neuronów, otrzymany wynik może być różne z uwagi na różne wagi w tych neuronach (Rysunek 4.10).



Rysunek 4.10 Schemat działania wag w neuronach

C. Funkcje aktywacji

Następnym krokiem po agregacji danych wejściowych (Rysunek 4.11), z zastosowaniem wag znajdujących się w neuronie, jest powstanie sygnału sumarycznego pobudzenia.

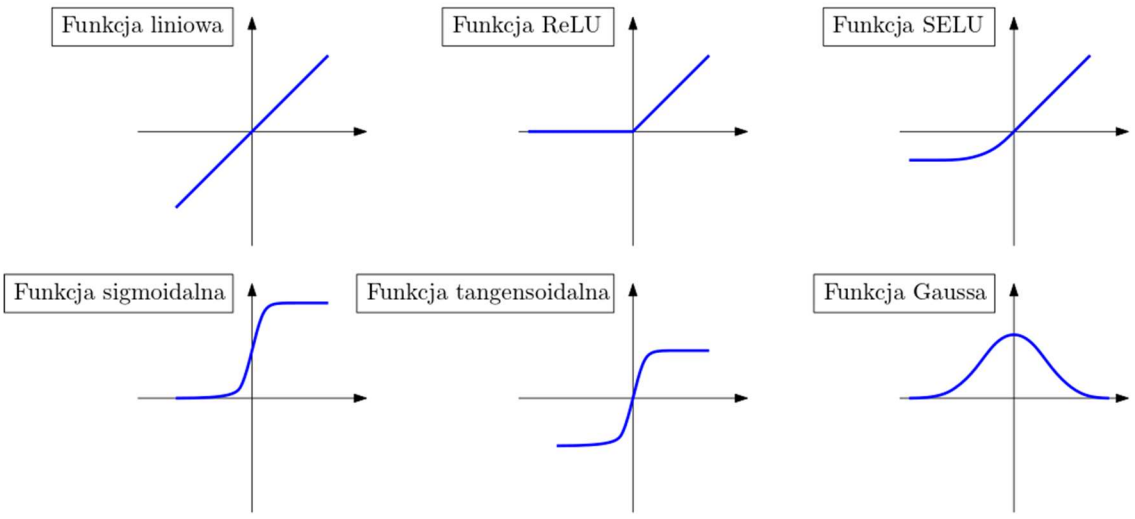


Rysunek 4.11 Schemat procesu agregacji danych wejściowych w neuronie

Zadaniem funkcji aktywacji jest określenie sposobu wyznaczania wartości sygnału wyjściowego neuronu w oparciu o wartości tego sumarycznego pobudzenia. Najczęściej wykorzystywane funkcje aktywacji to:

- funkcja liniowa – neuron liniowy,
- funkcja sigmoidalna – neuron sigmoidalny,
- funkcja tangensoidalna – neuron tangensoidalny (jest to w istocie tangens hiperboliczny),
- funkcja Gaussa – neuron radialny,
- funkcja ReLU,
- funkcja SELU.

Rysunek 4.12 przedstawia wymienione powyżej funkcje.

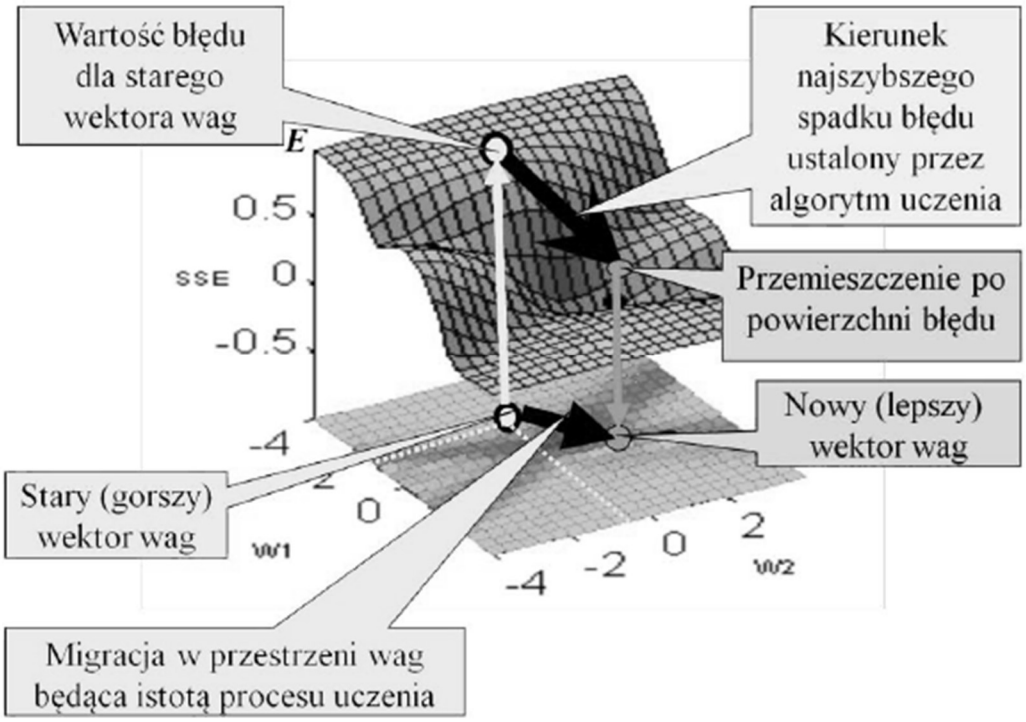


Rysunek 4.12 Zestawienie najczęściej stosowanych funkcji aktywacji

Zastosowanie ma również, nieuwzględniona na przedstawionej ilustracji, funkcja hiperboliczna (uzyskiwana wartość jest odwrotnością argumentu). Jest ona używana w neuronach Kohonena.

D. Proces uczenia

Proces ten polega na aktualizacji wag we wszystkich neuronach, przy użyciu algorytmu uczenia, poprzez dążenie do osiągnięcia minimum funkcji błędu, najczęściej przy użyciu gradientu tej funkcji. Funkcja ta określa wartość błędu popełnianego przez sieć neuronową (Rysunek 4.13). Błąd jest bezpośrednio zależny od dobranych współczynników wag. Proces działania algorytmu uczenia polega na iteracyjnym modyfikowaniu wag w neuronach.



Rysunek 4.13 Schemat poszukiwania minimum funkcji błędu w procesie uczenia

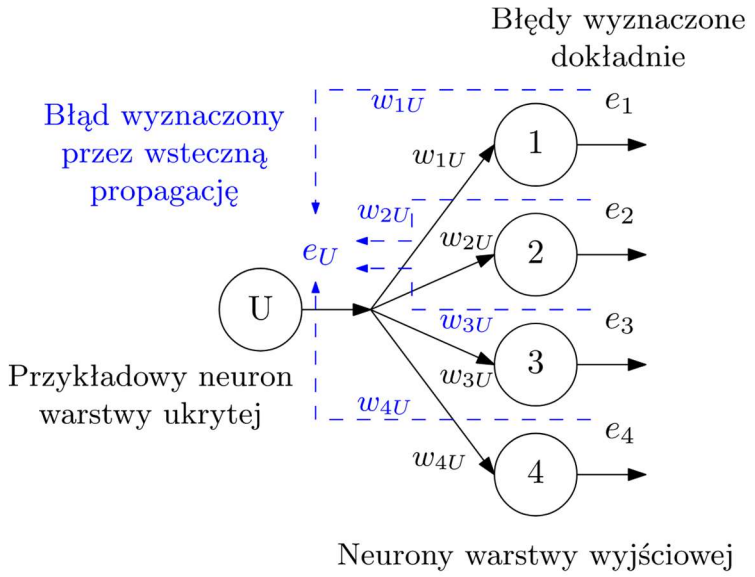
Istotnym aspektem procesu uczenia jest potrzeba znalezienia globalnego minimum funkcji błędu, uwzględniając fakt, że na przeszkodzie odpowiedniego dopasowania modelu mogą stanąć lokalne minima tej funkcji.

W procesie uczenia niezwykle istotny jest również współczynnik uczenia oraz ewentualnie bezwładność uczenia. Zatrzymanie algorytmu uczenia następuje w chwili wykorzystania zbioru walidacyjnego. Sygnalizuje to moment, w którym sieć zaczyna tracić zdolność generalizacji wyników uczenia. Po przebycie odpowiedniej ilości kroków uczenia, przygotowany model sieci neuronowej jest gotowy do przetestowania z udziałem zbioru danych przeznaczonych tylko do tego procesu (model po raz pierwszy przetworzy taki zestaw danych).

E. Propagacja wsteczna

Celem propagacji wstecznej jest wyznaczenie wartości błędów dla neuronów znajdujących się w warstwach ukrytych. Algorytm propagacji wstecznej (ang. backpropagation) , będący najstarszym algorytmem uczenia sieci neuronowych, bywa wykorzystywany do uczenia różnego typu sieci jednokierunkowych. Polega on na poprawianiu, na każdym kroku procesu uczenia, wartości korekty wag, bazując na ocenie błędu jaki jest popełniany przez każdy neuron w procesie uczenia sieci.

Stosowanie wstecznej propagacji błędu jest koniecznie z uwagi na to, że tylko błędy w neuronach warstwy wyjściowej wyznacza się bezpośrednio na podstawie danych wyjściowych i odpowiedzi wzorcowych zawartych w zbiorze uczącym. Z tego powodu, celem wyznaczenia błędu w neuronach warstw ukrytych wykorzystuje się opisywany algorytm. Przy wstecznej propagacji rozważany neuron otrzymuje (jest do niego przypisana) wartość błędu wyliczaną na podstawie wartości błędów wszystkich tych neuronów, do których wysyłał on wartość swojego sygnału wyjściowego jako składnika ich danych wejściowych Przy obliczaniu wartości wstecznie rzutowanego błędu uwzględnia się wartości wag połączeń pomiędzy rozważanym neuronem i neuronami, których błędy są do niego wstecznie rzutowane (przeciwnie do kierunku przepływu sygnału w jednokierunkowej sieci) (Rysunek 4.14).



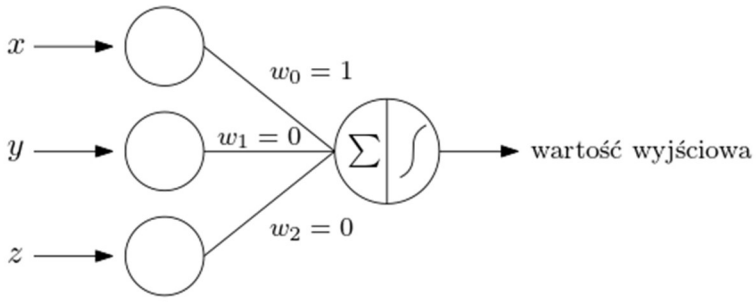
Rysunek 4.14 Uproszczony schemat działania propagacji wstecznej

Po obliczeniu wartości błędów neuronów, warstwy ukrytej najbliższej wyjścia, procedurę się powtarza, przyjmując wyliczone wartości błędów jako znane i w taki sam sposób przeprowadzając ich propagację do kolejnej warstwy ukrytej, bliższej wejścia.

Istnieje bardzo wiele typów sieci neuronowych. Wśród nich wyróżnić można siedem najczęściej stosowanych rodzajów. Są one następujące.

I. Perceptron

Pierwszym stworzonym, a zarazem podstawowym, rodzajem sieci neuronowej jest perceptron. W jego skład wchodzi zaledwie jeden neuron. Pobiera on wartości ze zbioru danych wejściowych, a następnie, po zsumowaniu wartości wejściowych przemnożonych przez odpowiednie wagi, aplikuje funkcję aktywacji, celem wygenerowania binarnej wartości wyjściowej (Rysunek 4.15). Ten typ sieci nie zawiera żadnych ukrytych warstw, a zatem nie można go zaklasyfikować do metod DL. Można używać go wyłącznie do zadań związanych z klasyfikacją binarną.

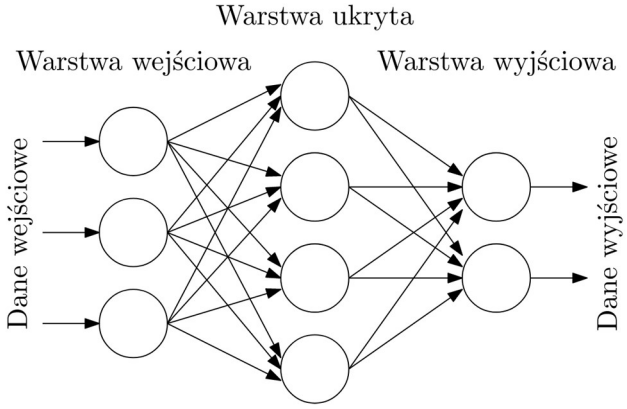


Rysunek 4.15 Schemat perceptronu

II. Jednokierunkowe sieci neuronowe

Jednokierunkowe sieci neuronowe (ang. Feedforward Neural Network; FNN) są złożone z dużej ilości połączonych ze sobą neuronów oraz warstw ukrytych. Nazwa tego typu sieci nawiązuje do sposobu przepływu danych przez sieć. Przepływ danych odbywa się tylko do przodu, w przypadku tego typu sieci nie stosuje się propagacji wstecznej. Stosowanie warstw ukrytych jest opcjonalne (Rysunek 4.16).

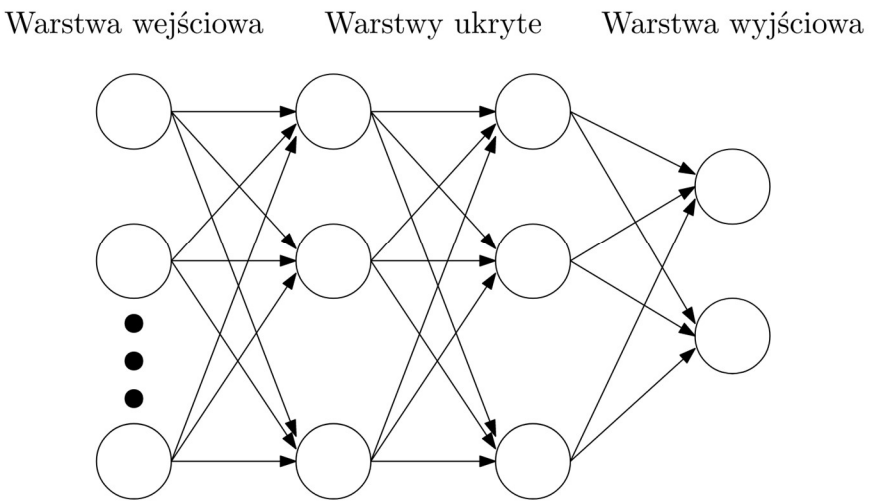
FNN wykorzystuje się przeważnie w zagadnieniach związanych z klasyfikacją, rozpoznawaniem mowy, twarzy bądź wzorców.



Rysunek 4.16 Schemat jednokierunkowej sieci neuronowej

III. Wielowarstwowy perceptron

Kolejnym typem sieci neuronowych są wielowarstwowe perceptrony (ang. Multi-Layer Perceptron; MLP). Ich główną przewagą względem FNN jest zastosowanie wstecznej propagacji w procesie uczenia. Sieci te składają się z wielu warstw ukrytych i funkcji aktywacji. Aktualizacja wag odbywa się za pomocą metody gradientu prostego (ang. gradient descent). MLP nazywane są również sieci gęstymi, z uwagi na to, że wszystkie neurony danej warstwy są połączone ze wszystkimi neuronami warstwy kolejnej (Rysunek 4.17).

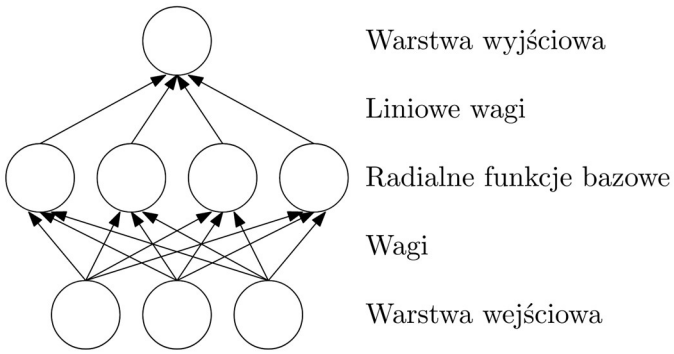


Rysunek 4.17 Schemat wielowarstwowego perceptronu

Tego typu sieci są najczęściej używane w aplikacjach opartych na metodach DL, lecz z uwagi na swoją gęstą i złożoną strukturę, uznaje się je niekiedy za mało efektywne – wolne.

IV. Radialne sieci neuronowe

Radialne sieci neuronowe (ang. Radial Basis Networks; RBN) w sposób odmienny niż pozostałe typy sieci przewidują wartości wyjściowe. Składają się one z trzech warstw – warstwy wejściowej, warstwy z neuronami RBF (radial basis function – radialna funkcja bazowa) oraz warstwy wyjściowej. Różnica pomiędzy neuronem RBF, a wielowarstwowym perceptronem wynika z rodzaju zastosowanej funkcji aktywacji – funkcji promieniowej w przypadku neuronów RBF. Celem neuronów radialnych jest wychwycenie powtarzalnych cech zbiorów w danych wejściowych. Dany neuron radialny jest pobudzany gdy otrzymuje daną wejściową podobną do takiej, której nauczył się wcześniej jako pewnego reprezentanta pewnej cechy. W większości przypadków, warstwą wyjściową jest pojedynczy neuron (Rysunek 4.18).

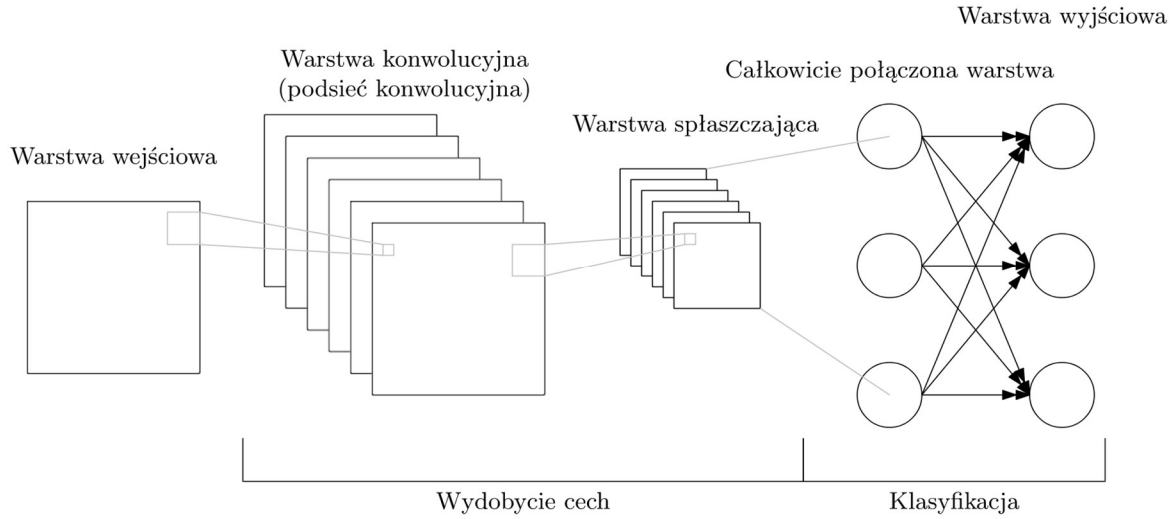


Rysunek 4.18 Schemat radialnej sieci neuronowej

Wykorzystuje się je przeważnie do aproksymacji funkcji w aplikacjach dla systemów, takich jak np. systemy przywracania mocy.

V. Konwolucyjne sieci neuronowe

Konwolujcyjne sieci neuronowe (ang. Convolutional Neural Networks – CNN) wykorzystuje się je głównie do zagadnień związanych z plikami graficznymi lub wideo. Ich najważniejszą zaletą jest sposób przetwarzania dużej ilości neuronów w warstwie wejściowej poprzez wychwytywanie, przy użyciu odpowiednich algorytmów (podsieci konwolucyjnej, zwanej niekiedy filtrami), najważniejszych cech zbioru danych wejściowych. W kolejnym kroku następuje proces spłaszczenia (ang. pooling), w którym zbiór danych przetwarzany jest do płaskiej, w pełni połączonej, warstwy. Rysunek 4.19 przedstawia uproszczony schemat CNN.

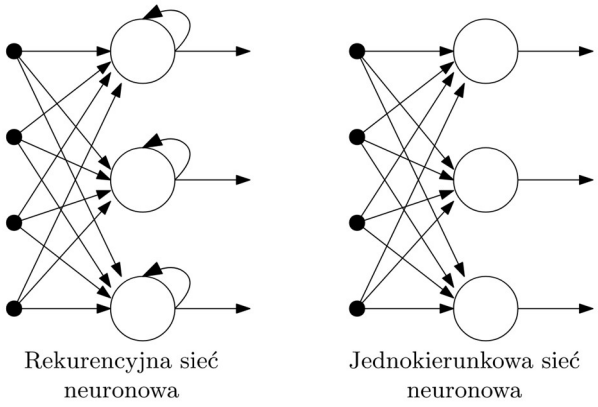


Rysunek 4.19 Uproszczony schemat konwolucyjnej sieci neuronowej

VI. Rekurencyjne sieci neuronowe

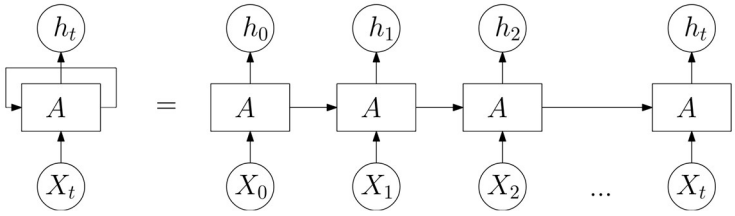
Rekurencyjne sieci neuronowe (ang. Recurrent Neural Networks – RNN) są najczęściej stosowane w zagadnieniach związanych z tekstem, dźwiękiem oraz z danymi uwzględniającymi rozkład czasowy. Są to tak zwane dane sekwencyjne.

Względem sieci jednokierunkowych, wyróżniają się połączeniami rekurencyjnymi w warstwach ukrytych (Rysunek 4.20).



Rysunek 4.20 Schematyczne porównanie jednokierunkowych oraz rekurencyjnych sieci neuronowych

W przypadku RNN wynik neuronu w danym kroku zależy od jego wyniku w kroku poprzednim (Rysunek 4.21).



Rysunek 4.21 Rozwinięcie schematu neuronu w przypadku rekurencyjnej sieci neuronowej

Główną wadą RNN jest problem zanikającego gradientu (ang. vanishing gradient), który zaburza proces zapamiętywania wag z wcześniejszych warstw

VII. Sieci pamięci krótkotrwałej

Sieci pamięci krótkotrwałej (ang. Long Short-Term Memory Networks; LSTM) są odpowiedzią na problem zanikającego gradientu w sieciach typu RNN. Są one wzbogacone o specjalną neuronową komórkę pamięci, w której informacja może być przechowywana przez długi czas. W sieciach typu LSTM wykorzystuje się tzw. „bramki" określające, które dane powinny zostać użyte, a które zapomniane. Rozróżnić można trzy rodzaje bramek – bramkę wejściową, bramkę wyjściową oraz bramkę zapominania. Bramka wejściowa określa, czy dane powinny być przechowywane w pamięci, bramka wyjściowa kontroluje dane przechodzące do kolejnej warstwy, natomiast bramka zapominania określa, czy dane, które nie są wymagane, mogą zostać zapomniane.

Sieci typu LSTM wykorzystuje się przeważnie do rozpoznawania gestów i mowy oraz do przewidywania tekstu.

4.4.2. Opis modelu

W tej pracy wykorzystana została głęboka sieć neuronowa z wsteczną propagacją, którą zaklasyfikować można do sieci typu MLP. Poniżej opisano schemat jej działania [40].

Poniższe równanie (4.14) opisuje wykorzystywany neuron nieliniowy:

$$y = \varphi(e), \tag{4.14}$$

gdzie φ jest wybraną nieliniową funkcją aktywacji. Sygnał e opisuje łączne pobudzenie neuronu (4.15) :

$$e = \sum_{i=0}^{n} w_i x_i, \tag{4.15}$$

gdzie sygnały $< x_1, x_2, \dots, x_n >$ składają się na wektor $\mathbf{X}$, natomiast w_0 jest składnikiem stałym neuronu, zatem $x_0 \equiv 1$.
Przyjmując $\mathbf{W}$ jako wektor wag $< w_1, w_2, \dots, w_n >$, sygnał e w postaci wektorowej można zapisać również następująco (4.16):

$$e = \mathbf{W}^{\mathbf{T}}\,\mathbf{X}\,. \tag{4.16}$$

Dla sieci wielowarstwowej założyć można ciąg uczący o następującej budowie (4.17):

$$\boldsymbol{U} = << \boldsymbol{X}^{(1)}, \boldsymbol{Z}^{(1)} >, < \boldsymbol{X}^{(2)}, \boldsymbol{Z}^{(2)} >, \dots, < \boldsymbol{X}^{(M)}, \boldsymbol{Z}^{(M)} >>. \tag{4.17}$$

Składa się on z par o postaci $< \boldsymbol{X}^{(j)}, \boldsymbol{Z}^{(j)} >$, które zawierają n-wymiarowe wektory $\boldsymbol{X}$ określone w j-tym kroku procesu uczenia oraz k-wymiarowe zestawy $\boldsymbol{Z}$ określające sygnały wyjściowe z elementów domykających sieć w tym kroku. Jeżeli wektor $\boldsymbol{Y}^{(j)}$ sygnałów wyjściowych z tych elementów nie będzie odpowiadał wymaganiom, czyli odnotowany zostanie błąd określony różnicą $\boldsymbol{Z}^{(j)} - \boldsymbol{Y}^{(j)}$, to ustalenie warstwy neuronów która odpowiada za ten błąd nie będzie możliwe. Rozwiązaniem tego problemu jest wykorzystanie algorytmu wstecznej propagacji błędów. Polega on na tym, że dysponując wyznaczonym błędem $\delta^{(m)(j)}$ (występującym podczas realizacji j-tego kroku procesu uczenia w neuronie o numerze m) istnieje możliwość przeniesienia tego błędu wstecz do wszystkich neuronów, których sygnały stanowiły wejścia dla m-tego neuronu.

Z uwagi na brak możliwości ścisłego rozgraniczenia sygnałów wejściowych i wyjściowych (sygnały wyjściowe jednej warstwy stają się sygnałami wejściowymi warstwy kolejnej) przyjęto jednolitą numerację neuronów (bez podziału na wejściowe, wyjściowe i należące do warstw ukrytych) oraz zastosowano stałe oznaczenie $y_m^{(j)}$ dla sygnału pojawiającego się na wyjściu m-tego neuronu w j-tym kroku procesu uczenia.

Ponadto, należy zauważyć, że wprowadzając jednolite oznaczenie również dla sygnałów wejściowych, każde wejście sieci można określić jako buforowane przez neuron, którego zadaniem jest przetworzenie wejściowego sygnału $x_i^{(j)} \in \boldsymbol{X}^{(j)}$ na sygnał $y_m^{(j)}$ (przy czym neuron ten ma charakter liniowy, a wartość wagi jedynego wejścia tego neuronu wynosi 1). Zależność tę określono poniższym równaniem (4.18):

$$y_m^{(j)} = x_i^{(j)}. \tag{4.18}$$

Można stwierdzić, że jedynym sposobem na rozróżnienie rodzaju danego sygnału (wejściowego, wyjściowego lub jednego z sygnałów warstwy ukrytej) jest określenie numeru m. Wprowadzono zatem zbiory numerów neuronów $\mathrm{M}(\forall_{\mathrm{m}} \in \mathrm{M})$ następujących podzbiorów:

$\mathrm{M_x}$ – zbiór numerów neuronów wejściowych do sieci;
M_y – zbiór numerów neuronów wyjściowych z sieci;
M_u – zbiór numerów neuronów warstwy ukrytej sieci;
$\mathrm{M_i}$ – zbiór numerów neuronów dostarczających sygnałów wejściowej do konkretnego, rozważanego aktualnie neuronu;
$\mathrm{M_o}$ – zbiór numerów neuronów do których dany, rozważany aktualnie neuron, wysyła swój sygnał wyjściowy.

Korzystając z powyższych oznaczeń można zatem opisać sygnał wyjściowy każdego m-tego neuronu sieci w j-tym kroku procesu uczenia (4.19):

$$y_m^{(j)} = \varphi\left(\sum_{i \in M_i} w_i^{(m)(j)} y_i^{(j)}\right), \tag{4.19}$$

gdzie $w_i^{(m)(j)}$ jest wartością współczynnik wagowego synapsy (połączenie dwóch neuronów) łączącej wejście m-tego neuronu z wyjściem i-tego neuronu, podczas j-tego kroku procesu uczenia. Kolejność obliczeń jest bezpośrednio związana z kolejnością przekazywania sygnałów z wejścia sieci do neuronów w kolejnych warstwach. Na początku należy wyznaczyć $y_m^{(j)}$ dla $m \in \mathrm{M}_x$, następnie dla $m \in \mathrm{M}_u$ (kolejno we wszystkich warstwach od wejścia w kierunku wyjścia), a na koniec wyznaczyć wartości dla $m \in \mathrm{M}_y$.

W przypadku metody wstecznej propagacji, określony porządek przy znajdowaniu wartości poprawionych wag synaptycznych jest dokładnie przeciwny do tego, jaki jest wymagany przy wyznaczaniu wartości sygnałów w kolejnych elementach sieci.

Pierwszym krokiem jest wyznaczenie poprawki dla neuronów stanowiących wyjściową warstwę sieci ($m \in \mathrm{M}_y$). Skoro dla poszczególnych sygnałów $y_m^{(j)}$ istnieją w ciągu uczącym oczekiwane wartości $z_m^{(j)}$, wyznaczenie błędu $\delta_m^{(j)}$ można dokonać

w sposób bezpośredni. Konieczne jest przy tym założenie, że numeracja składowych wektora $\boldsymbol{Z}^{(j)}$ jest jednakowa z numeracją neuronów tworzących wyjściową warstwę sieci. Wówczas:

$$\Delta w_i^{(m)(j)} = \eta\, \delta_m^{(j)} \frac{d\varphi(e)}{de_m^{(j)}}\, y_i^{(j)}, \tag{4.20}$$

gdzie dla $m \in \mathrm{M}_y$:

$$\delta_m^{(j)} = z_m^{(j)} - y_m^{(j)}. \tag{4.21}$$

Wzór ten dla $m \in \mathrm{M}_y$ oraz dla $\varphi(e)$ w postaci funkcji logistycznej może być zapisany w poniższej postaci (4.22):

$$\Delta w_i^{(m)(j)} = \eta\, \left(z_m^{(j)} - y_m^{(j)}\right)\left(1 - y_m^{(j)}\right)\, y_i^{(j)} y_m^{(j)}. \tag{4.22}$$

Stosując analogię, dla neuronów warstw ukrytych można określić poniższą regułę (4.23):

$$\Delta w_i^{(m)(j)} = \eta\, \delta_m^{(j)} \frac{d\varphi(e)}{de_m^{(j)}}\, y_i^{(j)}. \tag{4.23}$$

Jednak dla $m \in \mathrm{M}_u$ bezpośrednie określenie wartości $\delta_m^{(j)}$ nie jest możliwe.

Przy założeniu, że $M_o \subseteq M_y$ (mimo, że rozważany neuron należy do warstwy ukrytej to jego sygnał dociera wyłącznie do neuronów warstwy wyjściowej, czyli tych, dla których wartości błędów $\delta_m^{(j)}$ mogą zostać w łatwy sposób określone), można wykazać [36], że błąd $\delta_m^{(j)}$ neuronu warstwy ukrytej może być wyznaczony przez wsteczne rzutowanie (wsteczną propagację) błędów wykrytych w warstwie odbierającej sygnały:

$$\delta_m^{(j)} = \sum_{k \in \mathrm{M}_o} w_m^{(k)(j)}\, \delta_k^{(j)}, \tag{4.24}$$

gdzie czynnik $w_m^{(k)(j)}$ jest wagą występującą w neuronie o numerze k przy jego wejściu o numerze m, czyli odbierającym sygnał od aktualnie rozważanego neuronu. Oznacza to, że rzutowane wstecznie błędy mnożone są przez te same współczynniki, przez które mnożone były przesyłane sygnały, tyle tylko, że kierunek przesyłania informacji został w tym przypadku odwrócony.

Do stworzenia algorytmu DNN w języku Python wykorzystano następujące biblioteki: *torch* [41], *torchinfo* [42], *skorch* [43], *sklearn* [27], *openpyxl* [28], *pandas* [29], *numpy* [30], *itertools* [31] oraz *tqdm* [32]. Dla stworzonego modelu przeprowadzono

proces uczenia przy użyciu ustalonego wcześniej zbioru danych (punkt 4.2.2). Tablica 4.7 przedstawia przyjęte w modelu parametry wraz z ich opisem.

Tablica 4.7 Opis parametrów modelu predykcyjnego DNN

Parametr	**Opis parametru**
Ilość warstw ukrytych (ang. number of layers)	Ilość warstw znajdujących się między warstwą wejściową, a warstwą wyjściową.
Ilość neuronów w pojedynczej warstwie ukrytej (ang. number of hidden layers)	Ilość neuronów znajdujących się w każdej z przyjętych warstwę ukrytych.
Współczynnik uczenia (ang. learning rate)	Wartość zmniejszenia rozmiaru skoku, używana przy aktualizacji, aby zapobiec nadmiernemu dopasowaniu.
Funkcja aktywacji (ang. activation function)	Funkcja mająca na celu normalizację wartości wyjściowej neuronu.
Prawdopodobieństwo użycia wartości wyjściowej neuronu (ang. droput probability)	Wartość prawdopodobieństwa wykorzystania w dalszych obliczeniach wartości wyjściowej z pewnego neuronu w pewnej warstwie.
Użycie warstwy normalizacji (ang. batchnorm usage)	Zastosowanie dodatkowej warstwy ukrytej mającej na celu normalizację wartości wszystkich neuronów warstwy poprzedniej.
Współczynnik ograniczający złożoność modelu (ang. weight decay)	Współczynnik mający na celu ograniczyć złożoność stworzonego modelu i zapobiec zbytniemu dopasowaniu modelu do zbioru danych.
Wielkość paczki (ang. batchsize)	Ilość elementów ze zbioru danych wykorzystana w pojedynczej iteracji uczenia modelu (aktualizacji wag w neuronach).
Liczba epok (ang. epochs)	Ilość iteracji służących do wyznaczenia wag w neuronach pozwalających na osiągnięcie możliwe najmniejszego błędu dla przyjętego zbioru danych.

Celem znalezienia najbardziej optymalnych parametrów modelu predykcyjnego, przeanalizowano wiele kombinacji różnych wartości tych parametrów. Parametry końcowe wybrano celem uzyskania najmniejszego możliwego błędu średniokwadratowego (RMSE), który został określony równaniem (4.13). Tablica 4.8 przedstawia analizowane oraz finalnie przyjęte wartości omawianych parametrów.

Należy wspomnieć, że wszystkie dane w stworzonym zbiorze danych musiały zostać znormalizowane na potrzeby użycia algorytmu DNN. Spowodowane jest to niską efektywnością sieci neuronowych, w przypadku wartości wejściowych o różnym rzędzie wielkości. W związku z tym wykorzystano funkcję normalizującą z biblioteki *sklearn*.

Tablica 4.8 Przyjęte parametry modelu predykcyjnego DNN

Parametr	**Zakres wartości**	**Przyjęta wartość**
Ilość warstw ukrytych	5, 10, 20, 30	5
Ilość neuronów w pojedynczej warstwie ukrytej	5, 10, 25, 50, 100, 200	200
Współczynnik uczenia	0.1, 0.05, 0.01, 0.005, 0.001	0.001
Funkcja aktywacji	ReLU, Linear, SELU	Linear
Prawdopodobieństwo użycia wartości wyjściowej neuronu	0.1, 0.2, 0.3	0.0
Użycie warstwy normalizacji	True, False	False
Współczynnik ograniczający złożoność modelu	0.1, 0.01, 0.001, 0.0001	0.001
Wielkość paczki	32, 64, 128	64
Liczba epok	250, 500, 750, 1000	500

4.4.3. Wyniki

Błędy otrzymane dla przyjętych wartości parametrów modelu predykcyjnego metody DNN zaprezentowano poniżej (Tablica 4.9). Wyznaczono je zarówno dla zbioru danych treningowych jak i testowych.

Tablica 4.9 Błędy otrzymane dla modelu predykcyjnego DNN

Zbiór danych	**RMSE [GPa]**
Dane treningowe	12.0708
Dane testowe	12.8255

W załączniku nr 3 znajduje się kod źródłowy modelu stworzonego przy użyciu algorytmu DNN.

4.5. Porównanie otrzymanych wyników

W poniższej tablicy (Tablica 4.10) przedstawiono wyniki uzyskane za pomocą modeli predykcyjnych XGB oraz DNN wraz z wartościami oczekiwanymi modułu Younga pilastrów oraz określoną względną różnicą otrzymanych wyników dla dziesięciu losowych wartości ze zbioru danych testowych.

Tablica 4.10 Zestawienie wyników otrzymanych przy użyciu XGB oraz NN

L.p.	**Model predykcyjny XGB [GPa]**	**Model predykcyjny DNN [GPa]**	**Wartość oczekiwana [GPa]**	**Błąd względny XGB [%]**	**Błąd względny DNN [%]**
1	199.99	205.36	210.00	5.00	2.26
2	209.98	204.27	220.00	4.77	7.70
3	219.98	206.16	210.00	4.75	1.86
4	207.47	203.69	190.00	9.20	7.20
5	207.89	204.44	210.00	1.01	2.72
6	203.77	203.93	190.00	7.25	7.33
7	215.10	205.68	190.00	13.21	8.25
8	200.00	206.85	190.00	5.26	8.87
9	205.74	203.29	200.00	2.87	1.65
10	200.07	205.38	190.00	5.30	8.10

5. Nieniszczące badania laboratoryjne z wykorzystaniem metod sztucznej inteligencji

5.1. Badania laboratoryjne

5.1.1. Opis badania

W ramach odrębnej pracy, której streszczenie można opublikowano w [14], przeprowadzono badania laboratoryjne dla siedmiu płyt typu VIG. Tablica 5.1 przedstawia ich parametry geometryczne oraz własności mechaniczne.

Tablica 5.1 Parametry geometryczne oraz własności mechaniczne badanych laboratoryjnie elementów

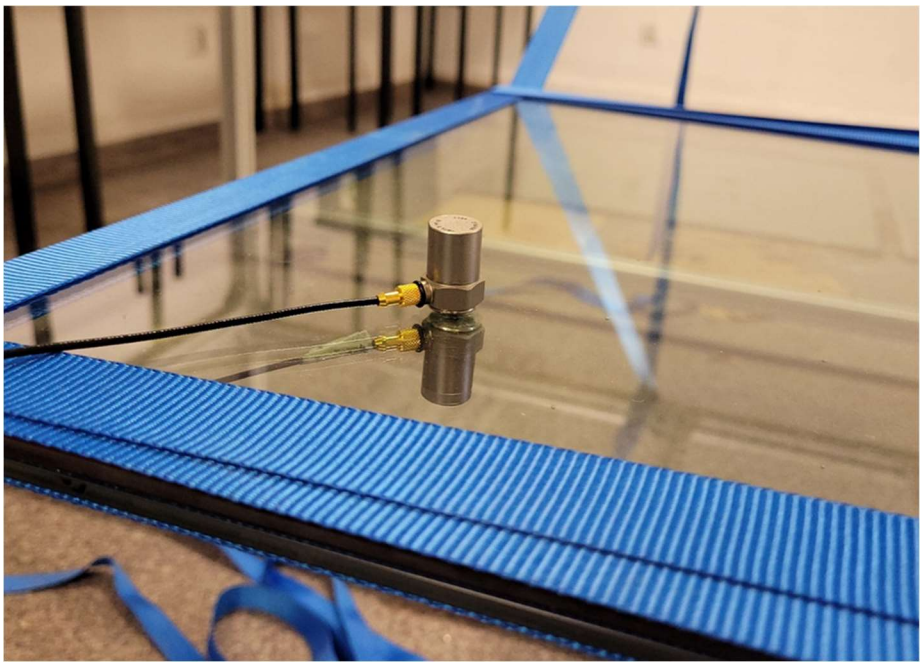
Parametr	Jednostka	Oznaczenie płyty						
		T1	T2	T3	T4	T5	T6	T7
A	m	0.40	0.50	0.50	1.00	1.00	1.00	1.00
B	m	0.80	0.60	0.60	1.00	1.00	1.00	1.50
t_g	mm	5	4	5	4	5	6	6
t_v	mm	0.30						
w_s	mm	9.0						
n_x	-	6	8	8	17	17	17	17
n_y	-	14	10	10	17	17	17	26
d_p	mm	0.60						
x_p	mm	62.5	57.5	57.5	60.0	60.0	60.0	60.0
y_p	mm	42.5	52.5	52.5	60.0	60.0	60.0	62.5
ρ_g	kg/m3	2500.0						
E_g	GPa	72.0						
ν_g	-	0.22						
Ep	GPa	200.0						
ρ_p	kg/m3	7850.0						
ν_p	-	0.31						
ρ_s	kg/m3	7850.0						
E_s	GPa	200.0						
ν_s	-	0.31						

W celu przetestowania okien próżniowych powstało specjalne stanowisko testowe w kształcie prostopadłościanu o długości boku około 2.00 m. Konstrukcja stanowiska wykonana jest z profili aluminiowych. Panele kompozytowe zostały zawieszone na aluminiowej ramie za pomocą elastycznych taśm z tkaniny. Założono, że taśma będzie owinięta wokół płyty wzdłuż wszystkich czterech krawędzi. Podstawowe właściwości mechaniczne wykorzystanych taśm elastycznych określono na podstawie dodatkowo przeprowadzonych badań (punkt 5.1.2). Rysunek 5.1 ilustruje sposób zawieszania paneli.

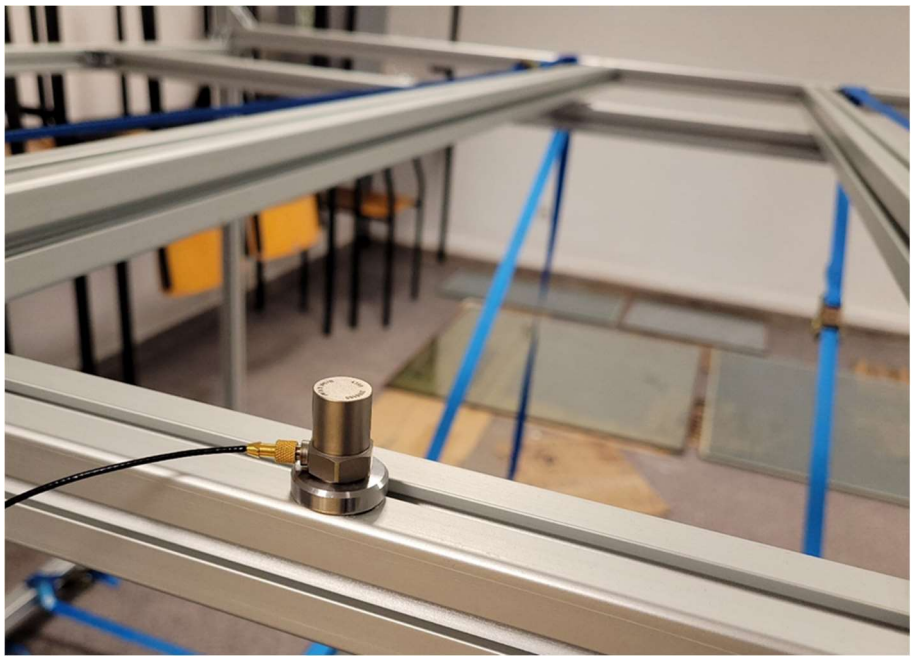
Rysunek 5.1 Sposób podparcia badanych elementów za pomocą elastycznych taśm

Do badania wykorzystano nieniszczącą technologię pomiarową opartą na metodzie badania wibracji PULSE firmy Brüel & Kjær. Jako urządzenie wzbudzające zastosowano specjalistyczny młot udarowy Brüel & Kjær typ 8206, natomiast odpowiedź dynamiczną badanych elementów mierzono za pomocą akcelerometrów Brüel & Kjær typ 4399 (Rysunek 5.2).

W celu uwzględnienia niepożądanego wpływu drgań ramy na otrzymywane wyniki, jednakowy (jak opisany wcześniej) akcelerometr zamontowano również do jednego z elementów aluminiowej ramy (Rysunek 5.3).



Rysunek 5.2 Sposób mocowania akcelerometru do badanego elementu



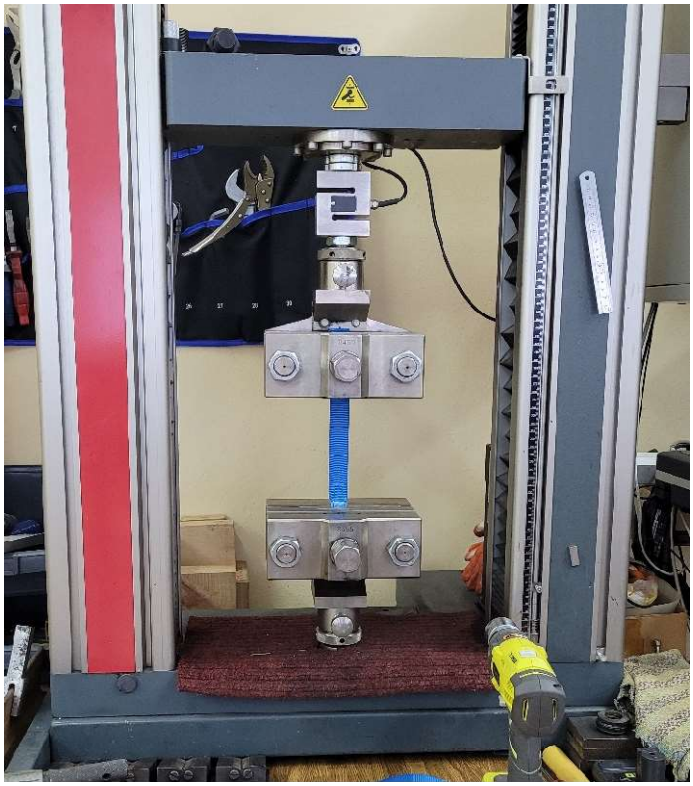
Rysunek 5.3 Sposób mocowania akcelerometru do aluminiowej ramy

Do przetworzenia wykonanych, za pomocą akcelerometrów, pomiarów paneli VIG zastosowano oprogramowanie Brüel & Kjær. Celem wyznaczenia poszukiwanych wartości częstotliwości drgań własnych, omawiane oprogramowanie wykorzystuje szybką transformatę Fouriera (FFT). Szybka transformata Fouriera to potężny algorytm do obliczania dyskretnej transformacji Fouriera (DFT).

Do określenia poszukiwanych częstotliwości drgań własnych wykorzystano wygenerowane w oprogramowaniu widma odpowiedzi dynamicznej płyt VIG uwzględniające odpowiedź dynamiczną aluminiowej konstrukcji ramy.

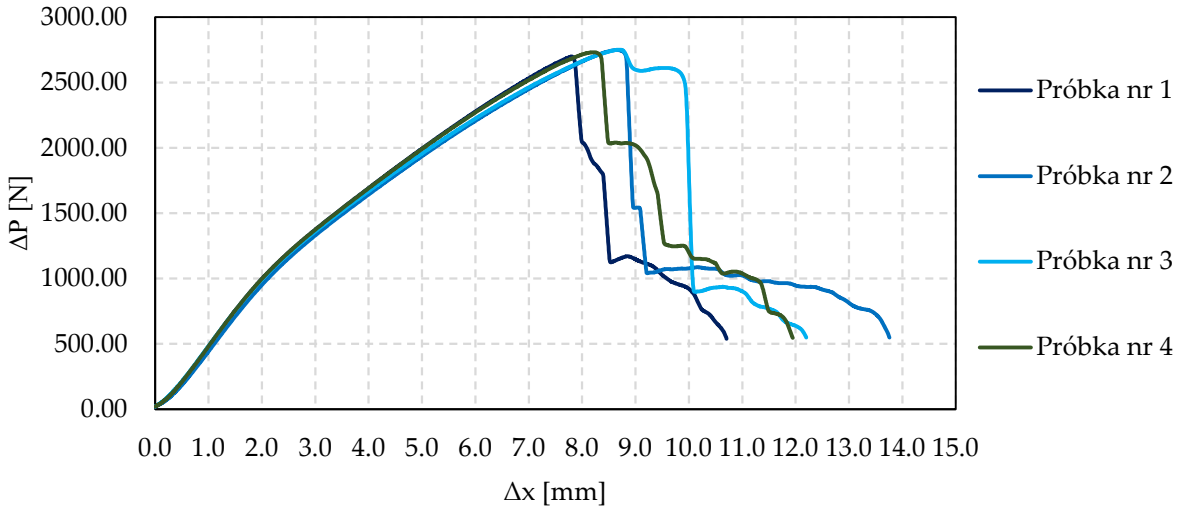
5.1.2. Warunki podparcia

W celu określenia modułu sprężystości podłużnej elastycznych taśm, przeprowadzono odpowiednie badania eksperymentalne. Cztery próbki taśmy poddano próbie jednoosiowego rozciągania (Rysunek 5.4).



Rysunek 5.4 Próba jednoosiowego rozciągania elastycznej taśmy

Długość próbek wynosiła 150 mm, szerokość 25 mm, a grubość 0.75 mm. Zależność siły od odkształcenia uzyskana na podstawie tych badań została przedstawiona poniżej (Rysunek 5.5).



Rysunek 5.5 Wykres zależności siły od przemieszczenia dla testowanych taśm

W oparciu o otrzymany wykres (Rysunek 5.5) określono moduł sprężystości podłużnej badanych taśm. Wartość tę wyznaczono jak w poniższym równaniu (5.1):

$$E = \frac{\sigma}{\varepsilon} = \frac{\frac{F}{A}}{\varepsilon} = \frac{\frac{1000\ N}{0.025\ m \cdot 0.00075\ m}}{\frac{0.002\ m}{0.150\ m}} = 4000000000.0\,\frac{N}{m^2} = 4.0\ GPa\ . \tag{5.1}$$

5.1.3. Wyniki eksperymentalne

W celu poprawnej analizy otrzymanych wyników stworzono modele numeryczne (punkt 5.2) dla parametrów geometrycznych i mechanicznych badanych okien typu VIG (Tablica 5.1), a następnie uzyskano częstotliwości drgań własnych. Uwzględniono otrzymane numerycznie postacie drgań własnych. Jest to bardzo istotne w przypadku płyt o wymiarach 1000 mm × 1000 mm, ponieważ częstotliwości własne dla niektórych dwóch kolejnych postaci drgań własnych są takie same. Wynika to z tych samych parametrów geometrycznych w dwóch kierunkach.
Tablica 5.2 przedstawia uśrednione wartości częstotliwości drgań własnych uzyskane ze wszystkich pomiarów.

Tablica 5.2 Wyniki przeprowadzonych badań laboratoryjnych

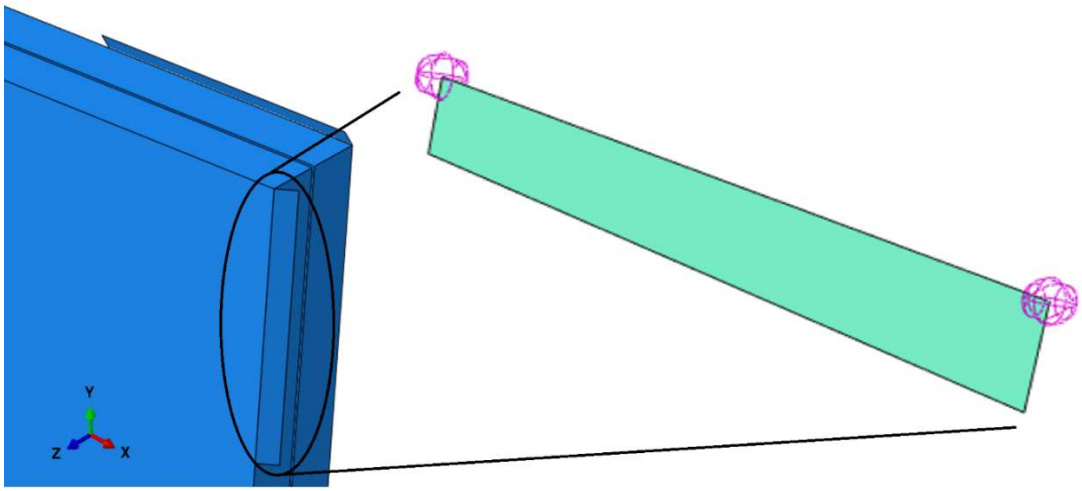
Postać drgań własnych	Częstotliwość drgań własnych dla płyty o danym oznaczeniu [Hz]						
	T1	T2	T3	T4	T5	T6	T7
1	16.0	19.0	15.0	11.0	11.0	10.0	5.0
2	20.0	21.0	17.0	11.0	11.0	10.0	10.0
3	26.0	26.0	24.0	13.0	15.0	12.0	14.0
4	40.0	40.0	39.0	14.0	17.0	17.0	18.0
5	45.0	50.0	49.0	24.0	24.0	25.0	24.0
6	55.0	54.0	50.0	24.0	24.0	25.0	27.0
7	95.0	115.0	136.0	39.0	52.0	58.0	47.0
8	126.0	123.0	140.0	51.0	64.0	63.0	57.0
9	195.0	163.0	190.0	64.0	74.0	79.0	68.0
10	230.0	214.0	255.0	81.0	95.0	104.0	73.0
11	275.0	235.0	279.0	81.0	95.0	104.0	78.0
12	326.0	262.0	305.0	114.0	120.0	141.0	90.0
13	340.0	339.0	410.0	114.0	120.0	141.0	110.0

14	343.0	345.0	420.0	124.0	144.0	165.0	120.0
15	436.0	354.0	428.0	128.0	149.0	179.0	129.0
16	475.0	415.0	506.0	150.0	160.0	184.0	146.0
17	483.0	440.0	533.0	168.0	195.0	244.0	156.0
18	589.0	504.0	611.0	177.0	204.0	247.0	160.0
19	594.0	517.0	625.0	177.0	204.0	247.0	174.0
20	635.0	563.0	685.0	188.0	225.0	260.0	194.0
21	674.0	586.0	724.0	209.0	254.0	279.0	205.0
22	680.0	648.0	789.0	209.0	254.0	279.0	225.0
23	759.0	673.0	815.0	234.0	285.0	336.0	227.0
24	766.0	689.0	831.0	247.0	291.0	345.0	245.0
25	858.0	694.0	842.0	250.0	303.0	356.0	247.0
26	889.0	775.0	849.0	268.0	320.0	364.0	259.0
27	903.0	803.0	991.0	268.0	320.0	364.0	265.0
28	956.0	870.0	1070.0	288.0	342.0	402.0	282.0
29	1051.0	902.0	1100.0	298.0	349.0	412.0	292.0
30	1090.0	913.0	1111.0	317.0	366.0	435.0	295.0

5.2. Model numeryczny

5.2.1. Opis modelu

Wykorzystano model numeryczny opisany w punkcie 2.3. Względem niego zmieniony został jedynie schemat statyczny rozpatrywanej płyty VIG. Został on przyjęty w oparciu o przeprowadzone badania eksperymentalne. Warunki brzegowe zostały odwzorowane numerycznie jako osiem małych elementów powłokowych na krótkich odcinkach krawędzi tafli szkła, w pobliżu naroży elementu VIG, obróconych o odpowiedni kąt w celu odwzorowania elastycznych taśm (Rysunek 5.6).



Rysunek 5.6 Numeryczne odwzorowanie elementów za pomocą elastycznych taśm

Sztywność tych drobnych elementów jest znacznie wyższa w porównaniu do sztywności szkła. Ich głównym celem było odwzorowanie kąta obrotu taśm podpierających. Te niewielkie elementy o wysokiej sztywności są punktowo podparte na dwóch jednokierunkowych podporach o skończonej sztywności (sprężynach). Sztywność pojedynczej sprężyny to całkowita sztywność liniowa jednego pasma podzielonego na dwie sprężyny węzłowe (5.2):

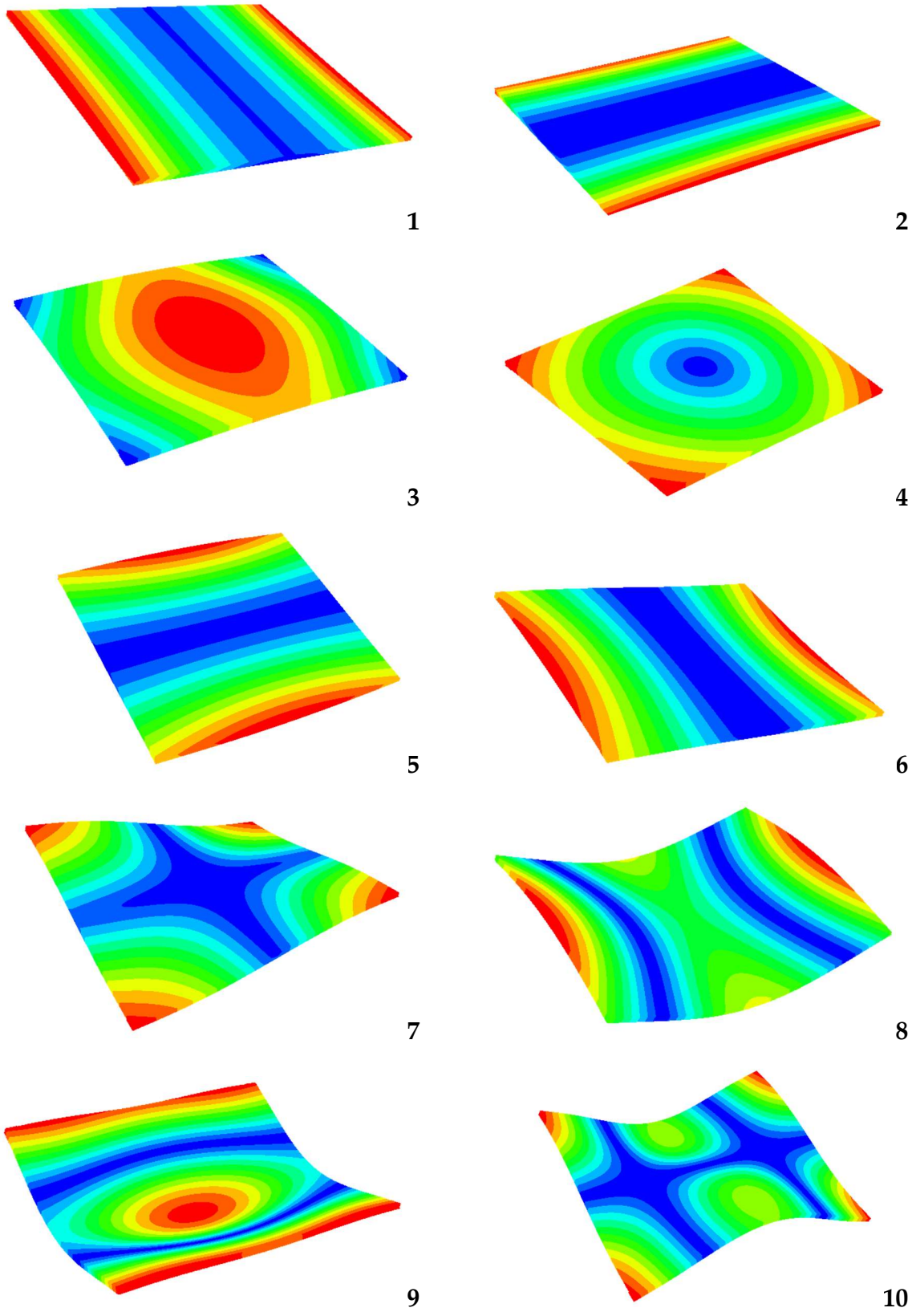
$$k_z = \frac{E \cdot A}{L_b} \cdot \frac{w_b}{w_b} \cdot 0.5 = \frac{E \cdot A}{L_b} \cdot 0.5 \tag{5.2}$$

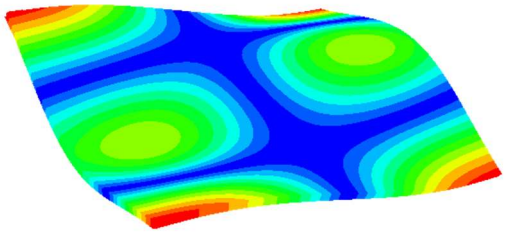
gdzie k_z jest jednokierunkową sztywnością węzłową, E jest modułem sprężystości podłużnej taśm, L_b jest długością taśmy podpierającej, A polem przekroju taśmy, a w_b jest szerokością taśmy.

Kod źródłowy algorytmu tworzącego modele numeryczne opisane powyżej znajduje się w załączniku nr 4.

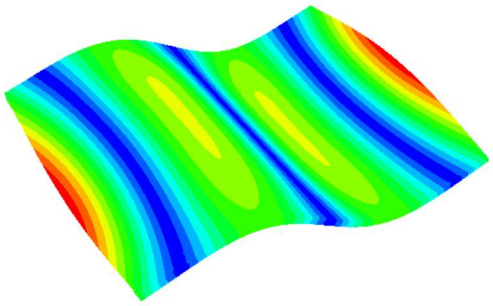
5.2.2. Wyniki analizy numerycznej

Wyznaczono trzydzieści pierwszych częstotliwości drgań własnych. Rysunek 5.7 przedstawia postacie drgań własnych dla otrzymanych częstotliwości drgań własnych dla płyty VIG o oznaczeniu T3.

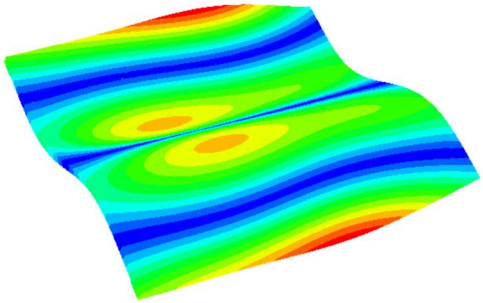




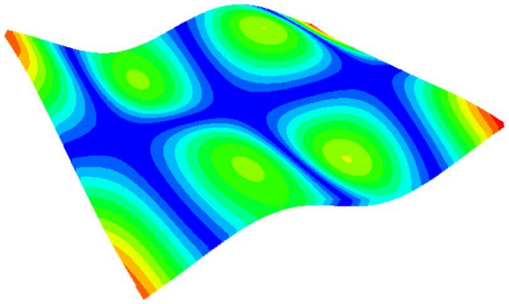
11



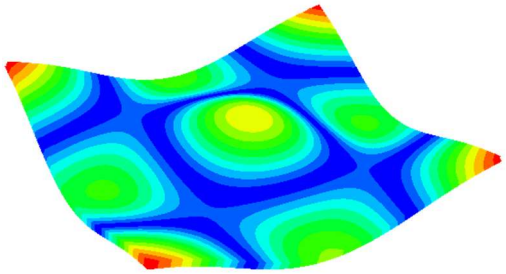
12



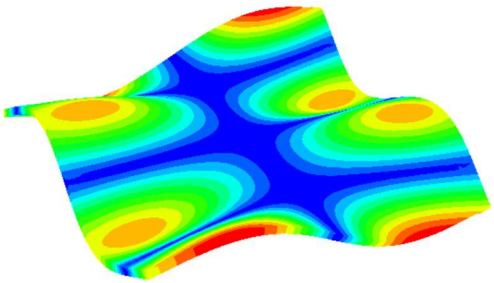
13



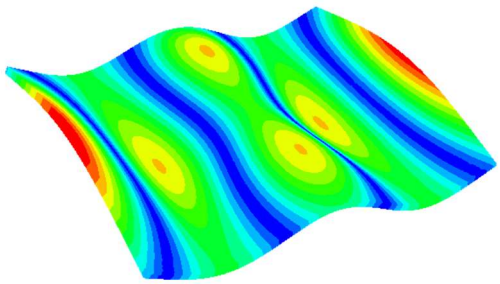
14



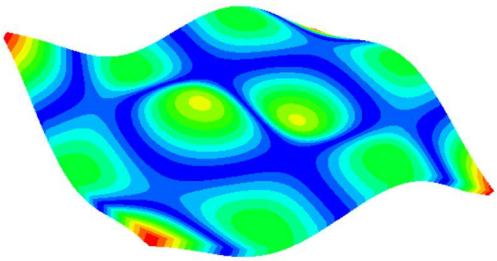
15



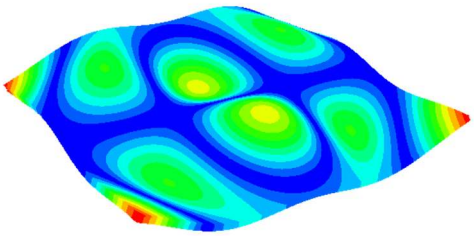
16



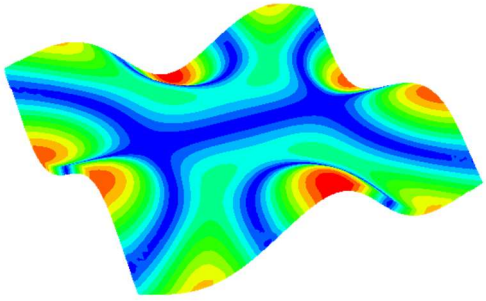
17



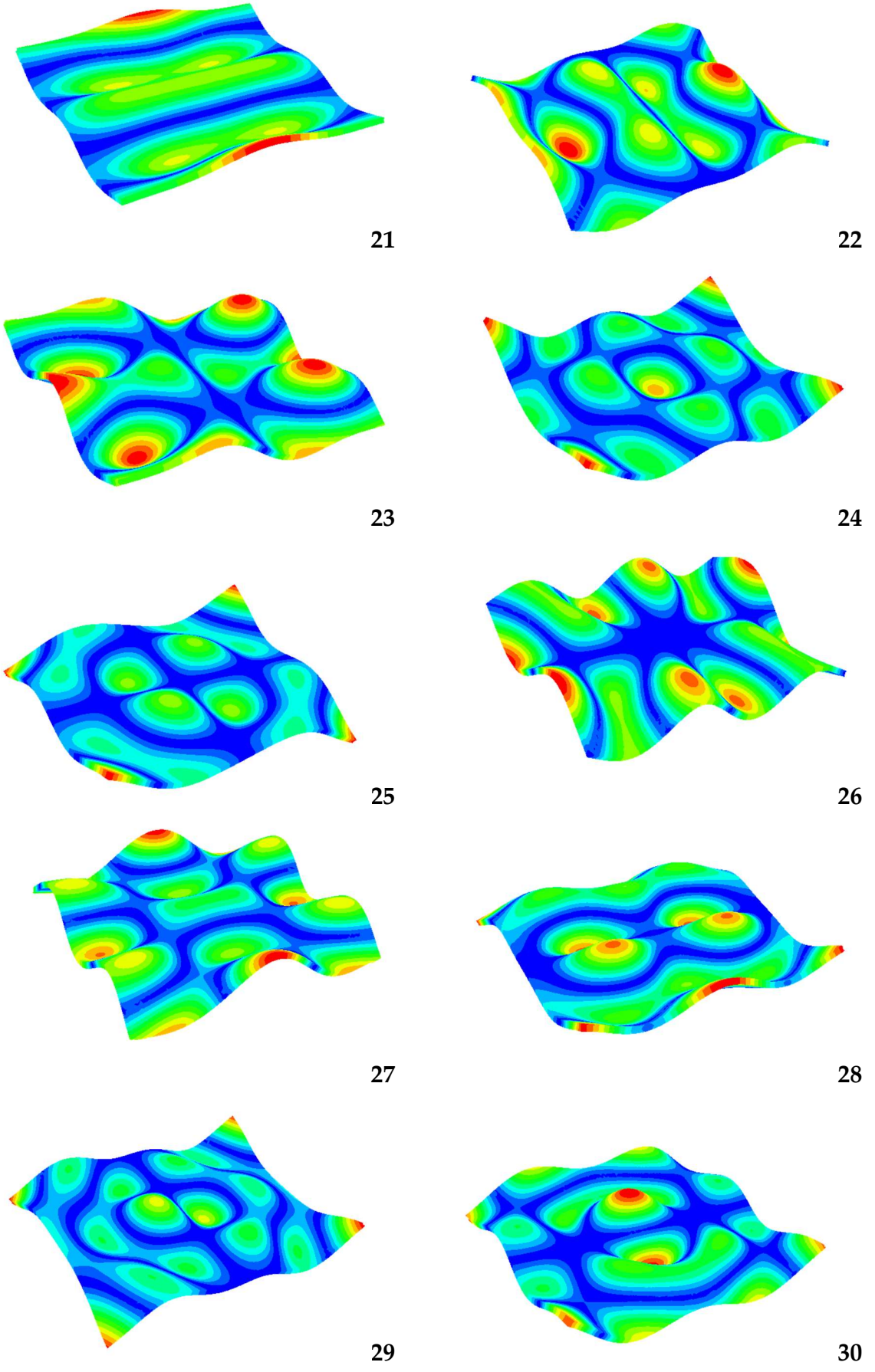
18



19



20



Rysunek 5.7 Postacie drgań własnych dla odpowiednio pierwszych trzydziestu częstotliwości drgań własnych

5.3. Sztuczna inteligencja

5.3.1. Zbiory danych

Celem stworzenia zbioru danych do trenowania modeli predykcyjnych określono finalnie przyjęte zmienne. (Tablica 5.3)

Tablica 5.3 Zestawienie finalnie przyjętych zmiennych w zbiorze danych

Oznaczenie parametru	Jednostka	Oznaczenie w zbiorze danych
DANE WEJŚCIOWE		
A	m	x_1
B	m	x_2
t_g	mm	x_3
n_x	-	x_5
n_y	-	x_6
x_p	mm	x_7
y_p	mm	x_8
$f_1 \div f_{30}$	Hz	$x_9 \div x_{38}$
DANE WYJŚCIOWE		
E_p	GPa	y

Wartości własne (częstotliwości drgań własnych) dla danego zbioru parametrów uzyskano za pomocą modeli numerycznych opisanych w punkcie 5.2. Tablica 5.4 przedstawia wszystkie wartości przyjęte do wykonania modeli numerycznych, na podstawie których wyznaczono wartości częstotliwości drgań własnych. Sumarycznie stworzono 315 różnych modeli numerycznych, poprzez wykonanie kombinacji uwzględniających wszystkie podane wartości.

Tablica 5.4 Zestawienie wartości wykorzystanych do stworzenia modeli numerycznych

Oznaczenie parametru	Jednostka	Przyjęte wartości	Ilość kombinacji
A	m	0.30; 0.60; 0.90; 1.20; 1.50	15
B	m	0.30; 0.60; 0.90; 1.20; 1.50	
t_g	mm	4.0; 5.0; 6.0	3
t_v	mm	0.30	
w_s	mm	9.0	1
n_x	-	Wartości uzależnione od wymiaru płyty: 0.30 – 5; 0.60 – 10; 0.90 – 15; 1.20 – 21; 1.50 – 21	-

n_y	-	Wartości uzależnione od wymiaru płyty: 0.30 – 5; 0.60 – 10; 0.90 – 15; 1.20 – 21; 1.50 – 21	
d_p	*mm*	0.6	1
x_p	*mm*	Wartości uzależnione od wymiaru płyty: 0.30 – 40.0; 0.60 – 52.5; 0.90 – 65.0; 1.20 – 50.0; 1.50 – 62.5	-
y_p	*mm*	Wartości uzależnione od wymiaru płyty: 0.30 – 40.0; 0.60 – 52.5; 0.90 – 65.0; 1.20 – 50.0; 1.50 – 62.5	
ρ_g	*kg/m³*	2500.0	1
E_g	*GPa*	72.0	1
ν_g	-	0.22	1
ρ_p	*kg/m³*	7850.0	1
ν_p	-	0.31	1
ρ_s	*kg/m³*	7850.0	1
E_s	*GPa*	210.0	1
ν_s	-	0.31	1
E_p	*GPa*	160.0; 170.0; 180.0; 190.0; 200.0; 210.0; 220.0	4
		Sumaryczna ilość wszystkich kombinacji	**315**

Zbiór danych został dokładnie opisany w punkcie 4.2. Cały zbiór danych został losowo podzielony na część treningową (80%) oraz testową (20%).

5.3.2. XGBoost

Stworzono model predykcyjny przy użyciu metody opisanej w punkcie 4.3. Tablica 5.5 przedstawia analizowane oraz finalnie przyjęte wartości omawianych parametrów.

Tablica 5.5 Parametry modelu predykcyjnego XGB

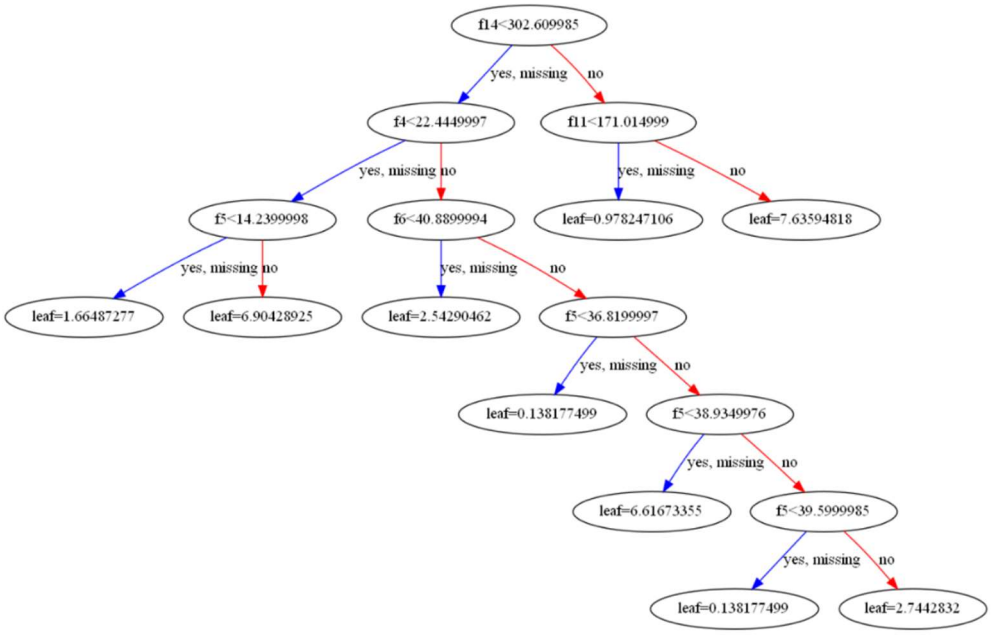
Parametr	**Zakres wartości**	**Przyjęta wartość**
Ilość estymatorów (drzew decyzyjnych)	500, 1000, 1500, 2000, 2500	2500
Współczynnik gamma	0.0, 0.1, 0.2	0.0
Współczynnik uczenia	0.01, 0.1, 0.2, 0.23, 0.26, 0.29, 0.32, 0.35	0.2
Maksymalna głębokość pojedynczego drzewa	3, 5, 10, 20, 30	10
Polityka wzrostu	Depthwise, Lossguide	Depthwise
Współczynnik podziału kolumny w drzewie	0.3, 0.6, 1.0	0.6
Współczynnik podziału podpróbek	0.3, 0.6, 1.0	0.6

Błędy otrzymane dla przyjętych wartości parametrów modelu predykcyjnego metody XGB zaprezentowano poniżej (Tablica 5.6). Wyznaczono je zarówno dla zbioru danych treningowych jak i testowych.

Tablica 5.6 Błędy otrzymane dla modelu predykcyjnego XGB

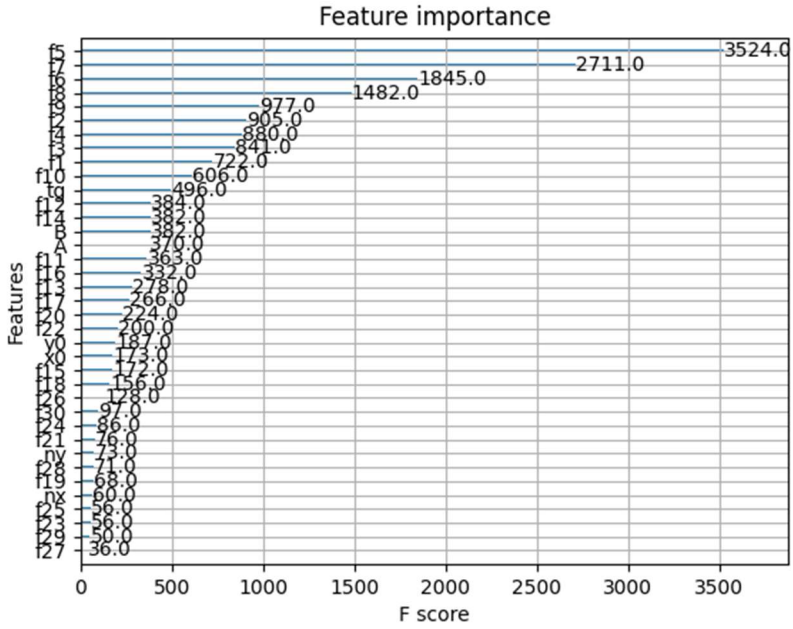
Zbiór danych	RMSE [GPa]
Dane treningowe	0.0005
Dane testowe	7.6107

Rysunek 5.8 przedstawia przykładowe drzewo decyzyjne ze zbioru wszystkich stworzonych automatycznie drzew decyzyjnych.



Rysunek 5.8 Przykładowe drzewo decyzyjne stworzonego modelu XGB

Rysunek 5.9 przedstawia wykres ważności cech (przyjętych zmiennych wejściowych) dla stworzonego modelu predykcyjnego metody XGB.



Rysunek 5.9 Wykres ważności cech dla stworzonego modelu XGB

5.3.3. Głęboka sieć neuronowa

Stworzono model predykcyjny przy użyciu metody opisanej w punkcie 4.4. Tablica 5.7 przedstawia analizowane oraz finalnie przyjęte wartości omawianych parametrów.

Tablica 5.7 Parametry modelu predykcyjnego DNN

Parametr	**Zakres wartości**	**Przyjęta wartość**
Ilość warstw ukrytych	5, 10, 20, 30, 50	5
Ilość neuronów w pojedynczej warstwie ukrytej	5, 10, 25, 50, 100	100
Współczynnik uczenia	0.1, 0.05, 0.01, 0.005, 0.001	0.1
Funkcja aktywacji	ReLU, Linear, SELU	SELU
Prawdopodobieństwo użycia wartości wyjściowej neuronu	0.1, 0.2, 0.3	0.0
Użycie warstwy normalizacji	True, False	True
Współczynnik ograniczający złożoność modelu	0.1, 0.01, 0.001, 0.0001	0.01
Wielkość paczki	32, 64, 128	64
Liczba epok	250, 500, 750, 1000	500

Błędy otrzymane dla przyjętych wartości parametrów modelu predykcyjnego metody DNN zaprezentowano poniżej (Tablica 5.6). Wyznaczono je zarówno dla zbioru danych treningowych jak i testowych.

Tablica 5.8 Błędy otrzymane dla modelu predykcyjnego DNN

Zbiór danych	**RMSE [GPa]**
Dane treningowe	20.4017
Dane testowe	18.6143

5.4. Wyniki

Najważniejszym punktem tego rozdziału jest wykorzystanie stworzonych modeli predykcyjnych do wyznaczenia modułu Younga pilastrów znajdujących się pomiędzy taflami szkła badanych płyt typu VIG. W tym przypadku, modele predykcyjne posłużyły jako funkcja określająca wartość modułu Younga, a wyniki eksperymentalne jako zmienne dla wspomnianej funkcji (Tablica 5.2). W oparciu o te założenia, poszukiwane wartości zostały określone przy użyciu dwóch modeli predykcyjnych dla siedmiu badanych szyb VIG. Tablica 5.9 przedstawia otrzymane wyniki.

Tablica 5.9 Wartości modułu Younga pilastrów wyznaczone przy użyciu stworzonych modeli predykcyjnych

Oznaczenie.	**Model predykcyjny XGB [GPa]**	**Model predykcyjny DNN [GPa]**	**Wartość oczekiwana [GPa]**	**Błąd względny XGB [%]**	**Błąd względny DNN [%]**
T1	208.65	193.52	200.00	4.33	3.35
T2	196.55	186.55	200.00	1.76	7.21
T3	212.32	192.91	200.00	6.16	3.68
T4	188.02	189.54	200.00	6.37	5.52
T5	182.54	192.88	200.00	9.57	3.69
T6	171.83	189.73	200.00	16.39	5.41
T7	185.71	189.80	200.00	7.69	5.37

6. Wnioski

W niniejszej pracy zbadano wpływ parametrów geometrycznych i materiałowych na odpowiedź dynamiczną płyt VIG. W celu uzyskania odpowiednich wartości wymaganych do dalszej analizy, stworzono numeryczny model powłokowo-prętowy w oprogramowaniu RFEM oraz numeryczny model trójwymiarowy w oprogramowaniu ABAQUS, a osiągnięte wyniki porównano. Rozważono różne rozmiary siatki elementów skończonych oraz różne sposoby numerycznego odwzorowania części składowych szyb próżniowych (szkło, pilastry, uszczelka). Jako lepszy sposób numerycznego odwzorowania omawianych szyb ustalono model trójwymiarowy stworzony przy użyciu oprogramowania ABAQUS CAE. W oparciu o wyniki uzyskane przy użyciu wybranego modelu przeprowadzono kolejne analizy.

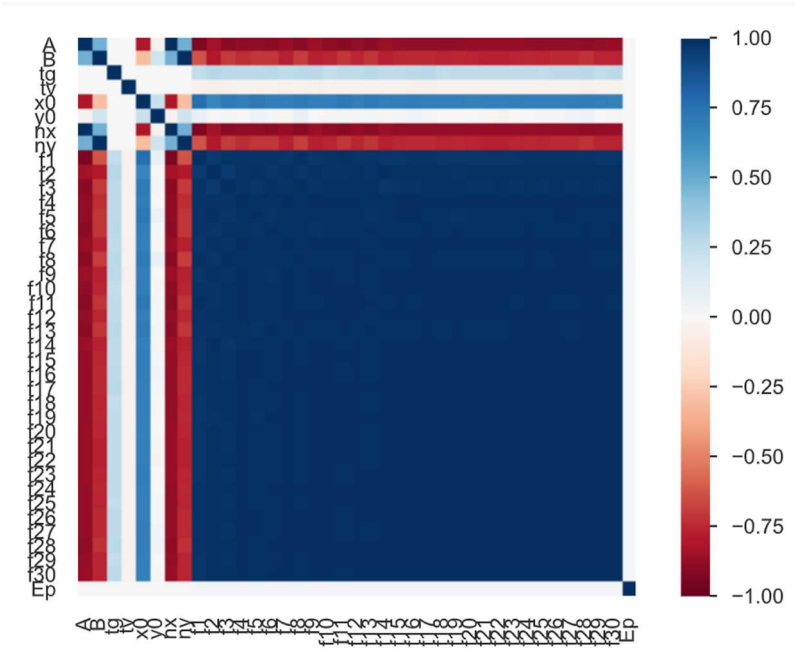
Ponadto, stworzono również model odwzorowujący numerycznie warunki laboratoryjne sposobu podparcia badanych płyt uwzględniające charakterystykę wytrzymałościową elastycznych taśm.

Zgodnie z otrzymanymi wynikami, właściwości mechaniczne i geometryczne analizowanego elementu mają wpływ na jego odpowiedź dynamiczną – częstotliwości drgań własnych. Zmiana modułu sprężystości podłużnej pilastrów powoduje nieznaczną zmianę wartości częstotliwości drgań własnych. Porównywalnie niewielki wpływ na tę wartość ma zmiana grubości warstwy próżni, natomiast zdecydowanie większe znaczenie w przypadku tej wartości ma zmiana grubości tafli szkła. Niemniej jednak, zmiana jakichkolwiek parametrów powoduje oczekiwaną zmianę wartości częstotliwości drgań własnych, a znormalizowane wartości częstotliwości drgań własnych względem dziesiątej częstotliwości drgań własnych dla płyty o ustalonych wymiarach zmieniają się o nieznaczną wartość.

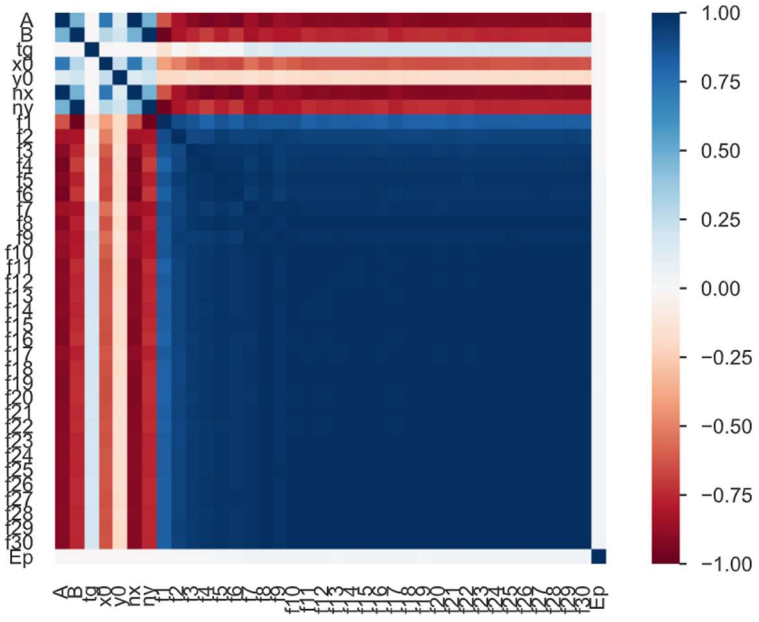
Innym ważnym aspektem szklenia próżniowego jest jego odpowiedź dynamiczna związana z drganiami w przeciw-fazie. Dotyczy to obu tafli szkła, z uwagi na szybko zmniejszającą się odległość między nimi podczas wspomnianego rodzaju drgań. Im mniejszy odstęp pomiędzy słupkami wsporczymi, a co za tym idzie sztywniejsza cała płyta VIG, tym większe prawdopodobieństwo wystąpienia wspomnianych drgań. Ze względu na niepożądany charakter tego zjawiska rozstaw słupów nośnych powinien być odpowiednio duży.

Kolejnym celem tej pracy była antycypacja modułu Younga pilastrów przy użyciu modeli predykcyjnych AI. Wykorzystano dwa różne typy modeli AI – Extreme Gradient Boosting (XGB) oraz Deep Neural Networks (DNN). Biorąc pod uwagę wyniki otrzymane zarówno dla zbioru danych określonego w rozdziale 4, jak i zbioru

danych określonego w rozdziale 5, pomimo braku znaczących wartości błędów względnych dla antycypowanych wartości (około 5%-10%), wyniki powinny mieć zdecydowanie mniejszy błąd i można je uznać za niezadowalające z uwagi na analizowany zakres wartości modułu Younga. Mimo, że zbiór danych jest dość mały, można zauważyć, że dane te są silnie skorelowane w przypadku obydwu różnych zbiorów danych. Rysunek 6.1 oraz Rysunek 6.2 przedstawiają, kolejno dla danych z rozdziału 4 oraz danych z rozdziału 5, wykresy współczynnika korelacji rang Spearmana określone dla wszystkich zmiennych względem siebie.



Rysunek 6.1 Wykres współczynnika korelacji rang Spearmana dla danych z rozdziału 4



Rysunek 6.2 Wykres współczynnika korelacji rang Spearmana dla danych z rozdziału 5

Współczynnik ten jest jedną z nieparametrycznych miar monotonicznej zależności statystycznej między zmiennymi. Przyjmuje on wartości z przedziału [-1, 1]. W odróżnieniu od współczynnika korelacji Pearsona, mierzącego poprawnie jedynie liniową zależność między zmiennymi, omawiany współczynnik uwzględnia dowolną monotoniczną zależność, w tym również nieliniową. Wykorzystuje się go do opisu siły korelacji dwóch cech. Wykresy współczynnika korelacji uzyskano przy użyciu biblioteki *pandas_profiling* [44].

W przypadku zbioru danych wykorzystanego w rozdziale 5 należy zauważyć mniejszą korelację zmiennych odpowiadających pierwszym czterem częstotliwościom drgań własnych (f_1; f_2; f_3; f_4). Wynika to bezpośrednio z charakteru postaci drgań powiązanych z tymi częstotliwościami. Postacie te odpowiadają za drgania szyby VIG, określonej w tym przypadku jako sztywna bryła, na elastycznych taśmach, wykorzystywanych jako podpory w warunkach laboratoryjnych. Pierwsza oraz druga są powiązane z postaciami przechyłowymi w dwóch kierunkach, trzecia z postacią, w której cała płyta przemieszcza się w kierunku normalnym do swej powierzchni, a czwarta z postacią obrotową (Rysunek 5.7). Można zatem stwierdzić, że pierwsze cztery częstotliwości drgań własnych w tak przyjętym schemacie statycznym ulegną zmianie tylko przy zmianie całkowitych wymiarów płyty. Wartości te nie są tym samym powiązane z modułem sprężystości podłużnej pilastrów, co dostrzec można również na wykresie ważności cech modelu predykcyjnego XGB (Rysunek 5.9).

Chociaż wyniki algorytmu XGB są nieco bardziej obiecujące niż wyniki algorytmu DNN, obydwa modele predykcyjne nie są w stanie dopasować się do rozwiązania ogólnego. Przewidywane wartości są zbliżone do średniej wartości całego zestawu danych. Wynika to prawdopodobnie z dwóch następujących aspektów: wpływ pilastrów na częstotliwości drgań własnych nie jest znaczący, a co za tym idzie, przyjęte dane nie mogą być znacząco zróżnicowane. Mając na uwadze istotność zbiorów danych w dziedzinie sztucznej inteligencji można wysnuć wniosek, że założenia dla zbiorów danych przyjętych w niniejszym opracowaniu są niewystarczające.

Istnieją dwa możliwe rozwiązania tego problemu. Pierwszym z nich jest znaczne rozszerzenie zbioru danych. Bardziej złożone dane mogą zdecydowanie zmniejszyć korelację zmiennych w zbiorach danych treningowych i testowych. Zamiast używać częstotliwości drgań własnych, można wykorzystać pełne widma odpowiedzi dynamicznej. Drugą możliwością jest wykorzystanie metod meta-uczenia się modeli algorytmach predykcyjnych [45]. Odnosi się to do zagadnienia, w którym algorytmy uczące uczą się od innych algorytmów uczących i stają się dzięki temu dużo bardziej wydajne. Wiąże się to również z doborem odpowiedniego modelu i dostrajaniem algorytmu.

Biorąc pod uwagę fakt, że badania nieniszczące płyt VIG mogą odegrać kluczową rolę w badaniu wpływu procesu starzenia się na trwałość tych okien, konieczne jest przeprowadzenie dalszych badań. Z tego powodu, przyszłe badania będą koncentrować się na rozwoju i walidacji dwóch wspomnianych metod.

Wyniki niniejszej pracy zostały zaprezentowane podczas konferencji „9th International Conference on Mechanics and Materials in Design" i opublikowane w materiałach z tej konferencji [46], [47].

Literatura

[1] A. Zoller Hohle Glasscheibe, Deutsches Reich Patents Nr. 387655, 1913

[2] T. Simko, "Heat transfer processes and stresses in vacuum glazing," The University of Sydney, Sidney, 1996.

[3] N. McSporran, "Properties and performance of vacuum insulated glazing," *Journal of Green Building*, vol. 9, no. 1, pp. 60–74, 2014, doi: 10.3992/1943-4618-9.1.60.

[4] C. Kocer, The Past, Present, and Future of the Vacuum Insulated Glazing Technology. Proceedings of Glass Performance Days 2019, https://www.glassonweb.com/article/past-present-and-future-vacuum-insulated-glazing-technology (dostęp: 03.07.2022).

[5] M. M. Vogel-Martin, M. B. Wolk, M. B. Free, O. Benson, E. L. Schwartz, R. F. Kamrath, B. U. Kolb, K. M. Humpal, M. J. Hendrickson Vacuum glazing pillars for insulated glass units, Application Number: 14/025958, United States Patent Application 20150079313, 2015

[6] L .Zhang, H. Zhang, A. Wang, Y. Zhao, Y. Li The Application and Development Trend of Vacuum Insulated Glass, The Glass Performance Days, 2019

[7] M. Teotia and R. K. Soni, "Applications of finite element modelling in failure analysis of laminated glass composites: A review," *Engineering Failure Analysis*, vol. 94, pp. 412–437, Dec. 2018, doi: 10.1016/J.ENGFAILANAL.2018.08.016.

[8] X. Centelles, F. Pelayo, M. Aenlle López, J. R. Castro, and L. F. Cabeza, "Long-term loading and recovery of a laminated glass slab with three different interlayers," *Scopus*, vol. 287, Jun. 2021, doi: 10.1016/J.CONBUILDMAT.2021.122991.

[9] C. Bedon, "Diagnostic analysis and dynamic identification of a glass suspension footbridge via on-site vibration experiments and FE numerical modelling," *Composite Structures*, vol. 216, pp. 366–378, May 2019, doi: 10.1016/J.COMPSTRUCT.2019.03.005.

[10] W. Zhu *et al.*, "Effects of pillar design on the thermal performance of vacuum-insulated glazing," *Construction and Building Materials*, vol. 316, Jan. 2022, doi: 10.1016/J.CONBUILDMAT.2021.125724.

[11] S. Cho and S.-H. Kim, "Analysis of the Performance of Vacuum Glazing in Office Buildings in Korea: Simulation and Experimental Studies" *Sustainability 2017, Vol. 9, Page 936*, vol. 9, no. 6, p. 936, Jun. 2017, doi: 10.3390/SU9060936.

[12] N. Ashmore, D. Cabrera, and C. Kocer, "Acoustic properties of vacuum insulating glazing" https://dspace.nal.gov.au/xmlui/bitstream/handle/123456789/402/p74.pdf?sequence=1 (dostęp: 04.07.2022)

[13] I. Kowalczyk, D. Kozanecki, S. Krasoń, and M. Rabenda, "Computational Modelling of VIG Plates Using FEM: Static and Dynamic Analysis," *Materials 2022, Vol. 15, Page 1467*, vol. 15, no. 4, p. 1467, Feb. 2022, doi: 10.3390/MA15041467.

[14] S. Krasoń, A. Wirowski, and Ł. Domagalski, "A non-destructive investigation of the dynamic properties of vig units," in *Proceedings M2D2022 - 9th International Conference on Mechanics and Materials in Design Funchal/Portugal 26-30 June 2022*, 2022.

[15] "Oprogramowanie do analizy statyczno-wytrzymałościowej | Dlubal Software." https://www.dlubal.com/pl (dostęp: 04.06.2022).

[16] "Elementy skończone w RFEM | Dlubal Software." https://www.dlubal.com/pl/pliki-do-pobrania-i-informacje/dokumenty/instrukcje-online/rfem-6/000434 (dostęp: 04.06.2022).

[17] "Abaqus Unified FEA - SIMULIA™ by Dassault Systèmes®." https://www.3ds.com/products-services/simulia/products/abaqus/ (dostęp: 04.06.2022).

[18] Z. Han, Y. Bao, W. Wu, Z. Liu, X. Liu, and Y. Tian, "Evaluation of thermal performance for vacuum glazing by using three-dimensional finite element model," *Key Engineering Materials*, vol. 492, pp. 328–332, 2012, doi: 10.4028/WWW.SCIENTIFIC.NET/KEM.492.328.

[19] "ABAQUS Version 6.6 Documentation." https://classes.engineering.wustl.edu/2009/spring/mase5513/abaqus/docs/v6.6/index.html (dostęp: 04.06.2022).

[20] "C3D8R and F3D8R." https://web.mit.edu/calculix_v2.7/CalculiX/ccx_2.7/doc/ccx/node27.html (dostęp: 04.06.2022).

[21] T. R. Sutter, C. J. Camarda, J. L. Walsh, and H. M. Adelman, "Comparison of several methods for calculating vibration mode shape derivatives," *AIAA Journal*, vol. 26, no. 12, pp. 1506–1511, 1988, doi: 10.2514/3.10070.

[22] B. Schäfer, D. Dörr, and L. Kärger, "Reduced-integrated 8-node hexahedral solid-shell element for the macroscopic forming simulation of continuous fibre-reinforced polymers," *Procedia Manufacturing*, vol. 47, pp. 134–139, 2020, doi: 10.1016/J.PROMFG.2020.04.154.

[23] A. Kaplan and M. Haenlein, "Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence," *Business Horizons*, vol. 62, no. 1, pp. 15–25, Jan. 2019, doi: 10.1016/J.BUSHOR.2018.08.004.

[24] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794. doi: 10.1145/2939672.2939785.

[25] B. Miller and L. Ziemiański, "Optimization of dynamic behavior of thin-walled laminated cylindrical shells by genetic algorithms and deep neural networks supported by modal shape identification," *Advances in Engineering Software*, vol. 147, p. 102830, Sep. 2020, doi: 10.1016/J.ADVENGSOFT.2020.102830.

[26] "XGBoost Documentation — xgboost 1.6.1 documentation." https://xgboost.readthedocs.io/en/stable/ (dostęp: 04.07.2022).

[27] "scikit-learn: machine learning in Python — scikit-learn 1.1.1 documentation." https://scikit-learn.org/stable/ (dostęp: 04.07.2022).

[28] "openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files — openpyxl 3.0.10 documentation." https://openpyxl.readthedocs.io/en/stable/ (dostęp: 04.07.2022).

[29] "pandas documentation — pandas 1.4.3 documentation." https://pandas.pydata.org/docs/ (dostęp: 04.07.2022).

[30] "NumPy Documentation." https://numpy.org/doc/ (dostęp: 04.07.2022).

[31] "itertools — Functions creating iterators for efficient looping — Python 3.10.5 documentation." https://docs.python.org/3/library/itertools.html (dostęp: 04.07.2022).

[32] "tqdm documentation." https://tqdm.github.io/ (dostęp: 04.07.2022).

[33] "Matplotlib — Visualization with Python." https://matplotlib.org/ (dostęp: 04.07.2022).

[34] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943, doi: 10.1007/BF02478259.

[35] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, Nov. 1958, doi: 10.1037/H0042519.

[36] P. Werbos and P. John, "Beyond regression : new tools for prediction and analysis in the behavioral sciences /," Jan. 1974.

[37] Y. LeCun *et al.*, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, Dec. 1989, doi: 10.1162/NECO.1989.1.4.541.

[38] M. Lefik, *Zastosowanie Metod Numerycznych W Analizie Wybranych Zagadnień Geotechniki I Geotechniki Środowiskowej*. Wydawnictwo Politechniki Łódzkiej, 2021.

[39] M. Szaleniec and R. Tadeusiewicz, "Leksykon sieci neuronowych [Lexicon on Neural Networks]," pp. 4–135, 2015.

[40] Tadeusiewicz R, *Sieci Neuronowe*. Akademicka Oficyna Wydawnicza, 1993.

[41] "PyTorch documentation — PyTorch 1.12 documentation." https://pytorch.org/docs/stable/index.html (dostęp: 04.07.2022).

[42] "torchinfo · PyPI." https://pypi.org/project/torchinfo/ (dostęp: 04.07.2022).

[43] "skorch documentation — skorch 0.11.0 documentation." https://skorch.readthedocs.io/en/stable/ (dostęp: 04.07.2022).

[44] "pandas_profiling API documentation." https://pandas-profiling.ydata.ai/docs/master/index.html (dostęp: 04.07.2022).

[45] J. Lee and C. Yang, "Deep neural network and meta-learning-based reactive sputtering with small data sample counts," *Journal of Manufacturing Systems*, vol. 62, pp. 703–717, Jan. 2022, doi: 10.1016/J.JMSY.2022.02.004.

[46] D. Kozanecki and M. Rabenda, "The analysis of vacuum glazing support pillars mechanical properties using neural networks," in *Proceedings M2D2022 - 9th International Conference on Mechanics and Materials in Design Funchal/Portugal 26-30 June 2022*, 2022.

[47] M. Rabenda and D. Kozanecki, "The analysis of vacuum glazing support pillars properties using extreme gradient boosting," in *Proceedings M2D2022 - 9th International Conference on Mechanics and Materials in Design Funchal/Portugal 26-30 June 2022*, 2022.

Wykaz rysunków

Wykaz tablic

Załączniki

Załącznik nr 1 – kod źródłowy skryptu 1 do tworzenia modeli w ABAQUS

```
from part import *
from material import *
from section import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from optimization import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *

#Models quantity
Mq = 360

class VIGmodel:
    def dimensions(self, A_dimension, B_dimension, thickness_glass,
thickness_vacuum, sealing_width):
        self.A  = A_dimension
        self.B  = B_dimension
        self.tg = thickness_glass
        self.tv = thickness_vacuum
        self.ws = sealing_width

    def pillarsgeom(self, x_quantity, y_quantity, diameter,
first_x_location, first_y_location):
        self.nx = x_quantity
        self.ny = y_quantity
        self.dp = diameter
        self.xp = first_x_location
        self.yp = first_y_location

    def glassprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_g = density
        self.E_g = modulus_elasticity
        self.v_g = friction_Poisson

    def pillarsprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_p = density
        self.E_p = modulus_elasticity
        self.v_p = friction_Poisson

    def sealingprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_s = density
        self.E_s = modulus_elasticity
        self.v_s = friction_Poisson

    def glassmesh(self, deviationFactor, minSizeFactor, size,
thickness_divide, finer_width_x, finer_width_y, finer_deviationFactor,
finer_minSizeFactor, finer_size):
        self.dF_g    = deviationFactor
```

```
        self.mSF_g   = minSizeFactor
        self.sz_g    = size
        self.tdv_g   = thickness_divide
        self.fw_x    = finer_width_x
        self.fw_y    = finer_width_y
        self.f_dF_g  = finer_deviationFactor
        self.f_mSF_g = finer_minSizeFactor
        self.f_sz_g  = finer_size

    def pillarsmesh(self, deviationFactor, minSizeFactor, size):
        self.dF_p  = deviationFactor
        self.mSF_p = minSizeFactor
        self.sz_p  = size

    def sealingmesh(self, deviationFactor, minSizeFactor, size,
thickness_divide):
        self.dF_s  = deviationFactor
        self.mSF_s = minSizeFactor
        self.sz_s  = size
        self.tdv_s = thickness_divide

    def create(self, model_num, eigenv_num, load_q):
        self.Mn = model_num
        self.En = eigenv_num
        self.Q  = load_q

        #Delete and create model
        mdb.Model(modelType=STANDARD_EXPLICIT, name='VIG' + str(self.Mn))

        #Create Glass1
        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=2.0)
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(0.0, 0.0),
point2=(self.A, self.B))
        mdb.models['VIG' + str(self.Mn)].Part(dimensionality=THREE_D,
name='Glass_1', type=DEFORMABLE_BODY)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].BaseSolidExtrude(depth=self.tg,
sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']

        #Create Glass2
        mdb.models['VIG' + str(self.Mn)].Part(name='Glass_2',
objectToCopy=mdb.models['VIG' + str(self.Mn)].parts['Glass_1'])

        #Create Sealing
        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=2.0)
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(0.0, 0.0),
point2=(self.A, self.B))
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(self.ws, self.ws),
point2=(self.A-self.ws, self.B-self.ws))
        mdb.models['VIG' + str(self.Mn)].Part(dimensionality=THREE_D,
name='Sealing', type=DEFORMABLE_BODY)
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].BaseSolidExtrude(depth=self.tv,
sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']

        #Change elements' names
```

```
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.DatumCsysByDefault(CARTESIAN)
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Glass_1-1',
part=mdb.models['VIG' + str(self.Mn)].parts['Glass_1'])
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Glass_2-1',
part=mdb.models['VIG' + str(self.Mn)].parts['Glass_2'])
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Sealing-1',
part=mdb.models['VIG' + str(self.Mn)].parts['Sealing'])

        #Create pillars
        Pillars = list()
        Pillars_cuts = list()
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=2.0)
                mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].CircleByCenterPerimeter(center=(self.
xp + (self.A-2*self.xp)/(self.nx-1)*i, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j), point1=(self.xp + (self.A-2*self.xp)/(self.nx-
1)*i + 0.5*self.dp, self.yp + (self.B-2*self.yp)/(self.ny-1)*j))
                mdb.models['VIG' +
str(self.Mn)].Part(dimensionality=THREE_D, name='Pillar_' + str(i) + '_' +
str(j), type=DEFORMABLE_BODY)
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].BaseSolidExtrude(depth=self.tv, sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'])
                del mdb.models['VIG' +
str(self.Mn)].sketches['__profile__']
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Pillar-' + str(i) +
'_' + str(j), part=mdb.models['VIG' + str(self.Mn)].parts['Pillar_' +
str(i) + '_' + str(j)])

                Pillars.append('Pillar-' + str(i) + '_' + str(j))
                Pillars_cuts.append(mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Pillar-' + str(i) + '_' + str(j)])

        #Pillars - second add and translate
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Pillar-' + str(i) +
'_' + str(j), part=mdb.models['VIG' + str(self.Mn)].parts['Pillar_' +
str(i) + '_' + str(j)])

        #Materials
        mdb.models['VIG' + str(self.Mn)].Material(name='Glass')
        mdb.models['VIG' +
str(self.Mn)].materials['Glass'].Density(table=((self.d_g, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Glass'].Elastic(table=((self.E_g, self.v_g), ))

        mdb.models['VIG' + str(self.Mn)].Material(name='Pillar')
        mdb.models['VIG' +
str(self.Mn)].materials['Pillar'].Density(table=((self.d_p, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Pillar'].Elastic(table=((self.E_p, self.v_p), ))

        mdb.models['VIG' + str(self.Mn)].Material(name='Sealing')
```

```
        mdb.models['VIG' +
str(self.Mn)].materials['Sealing'].Density(table=((self.d_s, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Sealing'].Elastic(table=((self.E_s, self.v_s), ))

        #Sections
        mdb.models['VIG' +
str(self.Mn)].HomogeneousSolidSection(material='Glass', name='Glass',
thickness=None)
        mdb.models['VIG' +
str(self.Mn)].HomogeneousSolidSection(material='Pillar', name='Pillar',
thickness=None)
        mdb.models['VIG' +
str(self.Mn)].HomogeneousSolidSection(material='Sealing', name='Sealing',
thickness=None)

        #Sections assignment
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].Set(cells=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].cells.getSequenceFromMask(('[#1 ]', ), ),
name='Set-1')
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].sets['Set-1'], sectionName='Glass',
thicknessAssignment=FROM_SECTION)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].Set(cells=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].cells.getSequenceFromMask(('[#1 ]', ), ),
name='Set-1')
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].sets['Set-1'], sectionName='Glass',
thicknessAssignment=FROM_SECTION)
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].Set(cells=mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].cells.getSequenceFromMask(('[#1 ]', ), ),
name='Set-1')
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].sets['Set-1'], sectionName='Sealing',
thicknessAssignment=FROM_SECTION)

        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].Set(cells=mdb.models['VIG' + str(self.Mn)].parts['Pillar_' +
str(i) + '_' + str(j)].cells.getSequenceFromMask(('[#1 ]', ), ), name='Set-
1')
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].SectionAssignment(offset=0.0, offsetField='',
offsetType=MIDDLE_SURFACE,
                    region=mdb.models['VIG' + str(self.Mn)].parts['Pillar_'
+ str(i) + '_' + str(j)].sets['Set-1'], sectionName='Pillar',
thicknessAssignment=FROM_SECTION)

        #Translations
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Glass_2-1', ),
vector=(0.0, 0.0, self.tg+self.tv))
```

```
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Sealing-1', ),
vector=(0.0, 0.0, self.tg))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=Pillars, vector=(0.0,
0.0, self.tg))

        #Surfaces
        mdb.models['VIG' + str(self.Mn)].rootAssembly.regenerate()
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass1_int',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, self.tg), )))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass1_ext',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, 0), )))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.regenerate()
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass2_int',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, self.tg + self.tv), )))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass2_ext',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, 2*self.tg + self.tv), )))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Sealing_int_g1',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Sealing-
1'].faces.findAt(((0.5*self.ws, 0.5*self.ws, self.tg), )))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Sealing_int_g2',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Sealing-
1'].faces.findAt(((0.5*self.ws, 0.5*self.ws, self.tg + self.tv), )))


        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Pillar_' + str(i) + '_' + str(j) +
'_int_g1', side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Pillar-' + str(i) + '_' +
str(j)].faces.getSequenceFromMask(('[#4 ]', ), ))
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Pillar_' + str(i) + '_' + str(j) +
'_int_g2', side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Pillar-' + str(i) + '_' +
str(j)].faces.getSequenceFromMask(('[#2 ]', ), ))


        #Ties
        mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass1_int'], name='Glass1_sealing',
positionToleranceMethod=COMPUTED, slave=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Sealing_int_g1'], thickness=ON,
tieRotations=ON)
```

```
        mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass2_int'], name='Glass2_sealing',
positionToleranceMethod=COMPUTED, slave=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Sealing_int_g2'], thickness=ON,
tieRotations=ON)

        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass1_int'], name='Glass1_pillar_' +
str(i) + '_' + str(j), positionToleranceMethod=COMPUTED,
slave=mdb.models['VIG' + str(self.Mn)].rootAssembly.surfaces['Pillar_' +
str(i) + '_' + str(j) + '_int_g1'], thickness=ON, tieRotations=ON)
                mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass2_int'], name='Glass2_pillar_' +
str(i) + '_' + str(j), positionToleranceMethod=COMPUTED,
slave=mdb.models['VIG' + str(self.Mn)].rootAssembly.surfaces['Pillar_' +
str(i) + '_' + str(j) + '_int_g2'], thickness=ON, tieRotations=ON)

        #Mesh
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].seedPart(deviationFactor=self.dF_g,
minSizeFactor=self.mSF_g, size=self.sz_g)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].seedEdgeByNumber(constraint=FINER,
edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].edges.getSequenceFromMask(('[#20 ]', ),),
number=self.tdv_g)

        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(gridSpacing=0.005,
name='__profile__',sheetSize=0.212,
            transform=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].MakeSketchTransform(sketchPlane=mdb.models['
VIG' + str(self.Mn)].parts['Glass_1'].faces[4], sketchPlaneSide=SIDE1,
            sketchUpEdge=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].edges[10], sketchOrientation=BOTTOM,
origin=(0.5*self.A, 0.5*self.B, self.tg)))
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].sketchOptions.setValues(decimalPlaces
=3)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].projectReferencesOntoSketch(filter=COPLANAR_
EDGES, sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        finer_edges_g1 = []
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x - 0.5*self.A, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j - 0.5*self.fw_y - 0.5*self.B), point2=(self.xp +
(self.A-2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x - 0.5*self.A, self.yp +
(self.B-2*self.yp)/(self.ny-1)*j + 0.5*self.fw_y - 0.5*self.B))
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
```

```
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j -
0.5*self.fw_y, self.tg))
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j +
0.5*self.fw_y, self.tg))
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].PartitionFaceBySketch(faces=mdb.models['VIG'
+ str(self.Mn)].parts['Glass_1'].faces.getSequenceFromMask(('[#10 ]', ),),
            sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'], sketchOrientation=BOTTOM,
sketchUpEdge=mdb.models['VIG' + str(self.Mn)].parts['Glass_1'].edges[10])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].seedEdgeBySize(constraint=FINER,
deviationFactor=self.f_dF_g, edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].edges.findAt(coordinates=tuple(finer_edges_g
1)), minSizeFactor=self.f_mSF_g, size=self.f_sz_g)
        mdb.models['VIG' + str(self.Mn)].parts['Glass_1'].generateMesh()


        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].seedPart(deviationFactor=self.dF_g,
minSizeFactor=self.mSF_g, size=self.sz_g)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].seedEdgeByNumber(constraint=FINER,
edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].edges.getSequenceFromMask(('[#20 ]', ),),
number=self.tdv_g)

        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(gridSpacing=0.005,
name='__profile__',sheetSize=0.212,
            transform=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].MakeSketchTransform(sketchPlane=mdb.models['
VIG' + str(self.Mn)].parts['Glass_2'].faces[4], sketchPlaneSide=SIDE1,
            sketchUpEdge=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].edges[10], sketchOrientation=BOTTOM,
origin=(0.5*self.A, 0.5*self.B, self.tg)))
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].sketchOptions.setValues(decimalPlaces
=3)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].projectReferencesOntoSketch(filter=COPLANAR_
EDGES, sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        finer_edges_g2 = []
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x - 0.5*self.A, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j - 0.5*self.fw_y - 0.5*self.B), point2=(self.xp +
(self.A-2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x - 0.5*self.A, self.yp +
(self.B-2*self.yp)/(self.ny-1)*j + 0.5*self.fw_y - 0.5*self.B))
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j -
0.5*self.fw_y, self.tg))
```

```
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j +
0.5*self.fw_y, self.tg))
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].PartitionFaceBySketch(faces=mdb.models['VIG'
+ str(self.Mn)].parts['Glass_2'].faces.getSequenceFromMask(('[#10 ]', ),),
            sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'], sketchOrientation=BOTTOM,
sketchUpEdge=mdb.models['VIG' + str(self.Mn)].parts['Glass_2'].edges[10])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].seedEdgeBySize(constraint=FINER,
deviationFactor=self.f_dF_g, edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].edges.findAt(coordinates=tuple(finer_edges_g
2)), minSizeFactor=self.f_mSF_g, size=self.f_sz_g)
        mdb.models['VIG' + str(self.Mn)].parts['Glass_2'].generateMesh()


        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].seedPart(deviationFactor=self.dF_s,
minSizeFactor=self.mSF_s, size=self.sz_s)
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].seedEdgeByNumber(constraint=FINER,
edges=mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].edges.getSequenceFromMask(('[#20 ]', ),),
number=self.tdv_s)
        mdb.models['VIG' + str(self.Mn)].parts['Sealing'].generateMesh()


        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].seedPart(deviationFactor=self.dF_p, minSizeFactor=self.mSF_p,
size=self.sz_p)
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].generateMesh()

        #BC
        faces_g1 = [(0, 0.5*self.B, 0.5*self.tg), (self.A, 0.5*self.B,
0.5*self.tg), (0.5*self.A, 0, 0.5*self.tg), (0.5*self.A, self.B,
0.5*self.tg)]
        faces_g2 = [(0, 0.5*self.B, 1.5*self.tg), (self.A, 0.5*self.B,
1.5*self.tg), (0.5*self.A, 0, 1.5*self.tg), (0.5*self.A, self.B,
1.5*self.tg)]

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Set(faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(coordinates=tuple(faces_g2))+mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(coordinates=tuple(faces_g1)), name='Set-1')

        #Initial step
        mdb.models['VIG' + str(self.Mn)].DisplacementBC(amplitude=UNSET,
createStepName='Initial', distributionType=UNIFORM, fieldName='',
localCsys=None, name='BC-1', region=mdb.models['VIG' +
str(self.Mn)].rootAssembly.sets['Set-1'], u1=SET, u2=SET, u3=SET,
ur1=UNSET, ur2=UNSET, ur3=UNSET)

        #Step - dynamics
        mdb.models['VIG' + str(self.Mn)].FrequencyStep(name='Step-1',
numEigen= self.En , previous='Initial')

        #Step - statics
```

```
        mdb.models['VIG' + str(self.Mn)].StaticStep(description='Static',
name='Step-2', previous='Step-1')
        mdb.models['VIG' + str(self.Mn)].steps['Step-
2'].setValues(initialInc=0.01)
        mdb.models['VIG' + str(self.Mn)].Pressure(amplitude=UNSET,
createStepName='Step-2', distributionType=UNIFORM, field='',
magnitude=self.Q, name='Vacuum1', region=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass1_ext'])
        mdb.models['VIG' + str(self.Mn)].Pressure(amplitude=UNSET,
createStepName='Step-2', distributionType=UNIFORM, field='',
magnitude=self.Q, name='Vacuum2', region=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass2_ext'])

        #Jobs
        mdb.Job(atTime=None, contactPrint=OFF, description='',
echoPrint=OFF, explicitPrecision=SINGLE, getMemoryFromAnalysis=True,
historyPrint=OFF,
            memory=90, memoryUnits=PERCENTAGE, model='VIG' + str(self.Mn),
modelPrint=OFF, name='Job-' + str(self.Mn), nodalOutputPrecision=SINGLE,
queue=None, resultsFormat=ODB,
            scratch='', type=ANALYSIS, userSubroutine='', waitHours=0,
waitMinutes=0)
        #mdb.jobs['VIG' + str(self.Mn)].setValues(numCpus=5, numDomains=5,
numGPUs=7)


        #Model regeneration
        mdb.models['VIG' + str(self.Mn)].rootAssembly.regenerate()

    def jobSubmit(self):
        mdb.jobs['Job-' + str(self.Mn)].submit(consistencyChecking=OFF)
        mdb.jobs['Job-' + str(self.Mn)].waitForCompletion()

    def results_save(self, model_num):
        self.Mn = model_num

        odb = openOdb(path='/home/grid/users/plgdamkoz/Job-' + str(self.Mn)
+ '.odb')
        frames=odb.steps['Step-1'].frames

        TxtVal=open("__1_EIGENv.txt", 'a')
        TxtFreq=open("__1_EIGENf.txt", 'a')

        TxtVal.write('Model_' + str(self.Mn) + ';')
        TxtFreq.write('Model_' + str(self.Mn) + ';')

        for frame in frames:
            f=frame.description
            if len(f[28:48])>1:
                TxtVal.write(str(float(f[24:37])) + ';')
                TxtFreq.write(str(float(f[44:57])) + ';')

        TxtVal.write('\n')
        TxtFreq.write('\n')

        TxtVal.close()
        TxtFreq.close()

        TxtDispl=open("__2_DISPL.txt", 'a')
        TxtTens=open("__2_TENS.txt", 'a')

        lastframe=odb.steps['Step-2'].frames[-1]
        displacements = lastframe.fieldOutputs['U'].values
        stresses = lastframe.fieldOutputs['S'].values
```

```
        displacementsL = []
        for v in displacements:
            displacementsL.append(v.data[2])

        TxtDispl.write('Model_' + str(self.Mn) + ';')
        TxtDispl.write(str(min(displacementsL)) + ';')
        TxtDispl.write(str(max(displacementsL)) + ';')

        stressesL = []
        for v in stresses:
            stressesL.append(v.mises)

        TxtTens.write('Model_' + str(self.Mn) + ';')
        TxtTens.write(str(min(stressesL)) + ';')
        TxtTens.write(str(max(stressesL)) + ';')

        TxtDispl.write('\n')
        TxtTens.write('\n')

        TxtDispl.close()
        TxtTens.close()

        odb.close()


#CREATING DATA
VIG_P = [[[0 for x in range(5)], [0 for x in range(5)], [0 for x in
range(3)], [0 for x in range(3)], [0 for x in range(3)]] for x in
range(Mq)]

for i in range(Mq):
    #DIMENSIONS
    #A
    VIG_P[i][0][0] = 0.6

    #B
    VIG_P[i][0][1] = 0.4

    #glass thickness
    VIG_P[i][0][2] = 0.004

    #vacuum thickness
    VIG_P[i][0][3] = 0.0003

    #sealing width
    VIG_P[i][0][4] = 0.009

    #PILLARSGEOM
    #quantity
    VIG_P[i][1][0] = 10

    VIG_P[i][1][1] = 6

    VIG_P[i][1][2] = 0.0006

    VIG_P[i][1][3] = 0.0525

    VIG_P[i][1][4] = 0.0625

    #GLASSPROP
    VIG_P[i][2][0] = 2500.0
    VIG_P[i][2][1] = 72.0*(10**9)
    VIG_P[i][2][2] = 0.22
```

```
    #PILLARSPROP
    VIG_P[i][3][0] = 7850.0
    VIG_P[i][3][2] = 0.31
    VIG_P[i][3][1] = 210.0*(10**9)

    #SEALINGPROP
    VIG_P[i][4][0] = 7850.0
    VIG_P[i][4][1] = 210.0*(10**9)
    VIG_P[i][4][2] = 0.31



#RUN CODE
VIGlist = list()

for i in range(Mq):
        VIGlist.append(VIGmodel())

for i in range(Mq):
    VIGlist[i].dimensions (VIG_P[i][0][0], VIG_P[i][0][1], VIG_P[i][0][2],
VIG_P[i][0][3], VIG_P[i][0][4])
    VIGlist[i].pillarsgeom(VIG_P[i][1][0], VIG_P[i][1][1], VIG_P[i][1][2],
VIG_P[i][1][3], VIG_P[i][1][4])

    VIGlist[i].glassprop  (VIG_P[i][2][0], VIG_P[i][2][1], VIG_P[i][2][2])
    VIGlist[i].pillarsprop(VIG_P[i][3][0], VIG_P[i][3][1], VIG_P[i][3][2])
    VIGlist[i].sealingprop(VIG_P[i][4][0], VIG_P[i][4][1], VIG_P[i][4][2])

    VIGlist[i].glassmesh(0.95, 0.95, 0.01, 2, 0.001, 0.001, 0.95, 0.95,
0.001)
    VIGlist[i].pillarsmesh(0.95, 0.95, 0.0001)
    VIGlist[i].sealingmesh(0.95, 0.95, 0.003, 3)

    VIGlist[i].create(i, 30, 100000)

for i in range(Mq):
    VIGlist[i].jobSubmit()
    VIGlist[i].results_save(i)
```

Załącznik nr 2 – kod źródłowy modelu XGBoost

```
import numpy as np
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import xgboost as xgb
import itertools
import pandas as pd
from tqdm import tqdm

# load the dataset
dataframe = read_csv('C:/Moje/Budownictwo/Politechnika Łódzka/II
STOPIEŃ/Praca Dyplomowa/Datasets/Obliczenia/data_P1_360.csv',
delimiter=';')
data = dataframe.values

orig_feature_names = list(dataframe.columns[:37])

# split data into input and output columns
X, y = data[:, :37], data[:, -1:]

# split data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)

def generate_parameter_combinations(search_space):
    keys = list(search_space.keys())
    values = [search_space[key] for key in keys]

    combinations = list(itertools.product(*values))
    return [{keys[i]: v for i, v in enumerate(c)} for c in combinations]


search_space = {
        "n_estimators": [500, 1000, 1500, 2000, 2500],
        "gamma" : [0.0, 0.1, 0.2],
        "learning_rate": [0.20, 0.23, 0.26, 0.29],
        "max_depth" : [3,5,10],#20,30],
        "grow_policy" : ['depthwise'], #'lossguide'],
        "colsample_bytree" : [0.3, 0.6, 1.0],
        "subsample" : [0.3, 0.6, 1.0],
}

all_parameters = generate_parameter_combinations(search_space)

search_space_result_df = pd.DataFrame()

for params in tqdm(all_parameters):
    row = params
    row["objective"] = 'reg:linear'

    model = xgb.XGBRegressor(**row)

    model.fit(X_train, y_train)

    y_pred1 = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred1))

    row["value"] = np.sqrt(mean_squared_error(y_test, y_pred1))
    search_space_result_df = search_space_result_df.append(row,
ignore_index=True)
```

```
search_space_result_df.to_csv("search_space_result_XGB.csv")
search_space_result_df.to_excel("search_space_result_XGB.xlsx")

# define model
model1 = xgb.XGBRegressor(objective = 'reg:linear', n_estimators = 500,
max_depth = 10, learning_rate = 0.2, colsample_bytree = 1, subsample = 1,
grow_policy = 'depthwise', gamma = 0.0)

# fit model
model1.fit(X_train, y_train)

model1.get_booster().feature_names = orig_feature_names

# compute error
y_pred1 = model1.predict(X_test)
y_pred2 = model1.predict(X_train)

rmse1 = np.sqrt(mean_squared_error(y_test, y_pred1))
rmse2 = np.sqrt(mean_squared_error(y_train, y_pred2))

print("RMSE test: %f" % (rmse1))
print("RMSE train: %f" % (rmse2))

#Plot
%matplotlib widget

import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [20, 20]
xgb.plot_tree(model1,num_trees=25)
plt.show()

plt.rcParams['figure.figsize'] = [20, 20]
xgb.plot_importance(model1)
plt.show()
```

Załącznik nr 3 – kod źródłowy modelu NN

```
from pandas import read_csv

#Dataset parametrese
dataset_path = "C:/Moje/Budownictwo/Politechnika Łódzka/II STOPIEŃ/Praca
Dyplomowa/Datasets/Obliczenia/data_P1_360.xlsx"
test_size = 0.2

# Training parameters
compression_method = None
feature_selection_method = None

random_state = 0

from sklearn.model_selection import train_test_split
import pandas as pd

df = pd.read_excel(dataset_path)

print(df.head())

y = df["Ep"]
del df["Ep"]
X = df

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=test_size, shuffle=True, random_state=random_state)

#Normalization
from sklearn import preprocessing
import numpy as np

normalizer = preprocessing.StandardScaler().fit(X_train)
X_train = normalizer.transform(X_train)
X_test = normalizer.transform(X_test)

#Create NN
import torch.nn as nn

def create_activation_function(activation_function):
    if activation_function == "relu":
        return nn.ReLU()
    elif activation_function == "linear":
        return nn.Identity()
    elif activation_function == "selu":
        return nn.SELU()

class Net(nn.Module):
    def __init__(self, input_size, output_size, number_of_layer,
number_of_hidden_layer_neurons, activation_function, dropout_probability,
use_batchnorm):
        super().__init__()

        modules = []

        prev_layer_size = input_size

        for i in range(number_of_layer):
            modules.append(nn.Sequential(
                nn.BatchNorm1d(prev_layer_size) if use_batchnorm else
nn.Identity(),
                nn.Dropout(dropout_probability),
```

```
                nn.Linear(prev_layer_size,number_of_hidden_layer_neurons),
                create_activation_function(activation_function)
            ))

            prev_layer_size = number_of_hidden_layer_neurons

        modules.append(nn.Linear(prev_layer_size, output_size))

        self.model = nn.Sequential(*modules)

    def forward(self, x):
        return self.model(x)

from skorch.net import NeuralNet
import torch

model = NeuralNet(
    module = Net,
    module__input_size=X_train.shape[1],
    module__output_size=1,

    module__number_of_layer=5,
    module__number_of_hidden_layer_neurons=200,
    module__activation_function = "linear",
    module__dropout_probability = 0.0,
    module__use_batchnorm = False,

    criterion=torch.nn.MSELoss,

    batch_size=64,
    max_epochs=500,

    optimizer=torch.optim.Adam,
    optimizer__lr=0.001,
    optimizer__weight_decay=0.001,

    device= 'cuda' if torch.cuda.is_available() else "cpu"
)

import itertools
from tqdm import tqdm

def generate_parameter_combinations(search_space):
    keys = list(search_space.keys())
    values = [search_space[key] for key in keys]

    combinations = list(itertools.product(*values))
    return [{keys[i]: v for i, v in enumerate(c)} for c in combinations]


search_space = {
    "number_of_layer" : [5, 10, 20, 30],
    "number_of_hidden_layer_neurons": [5, 10, 25, 50],
    "activation_function" : ["linear","relu", "selu"],
    "dropout_probability" : [0.1],
    "use_batchnorm" : [True],
    "lr" : [0.01, 0.05, 0.1],
    "weight_decay" : [0.01],
    "batch_size" : [32, 64, 128],
    "max_epochs" : [250, 500]

}

all_parameters = generate_parameter_combinations(search_space)
```

```
search_space_result_df = pd.DataFrame()

for i,params in tqdm(enumerate(all_parameters)):
    row = params

    try:
        model = NeuralNet(
            module = Net,
            module__input_size=X_train.shape[1],
            module__output_size=1,

            module__number_of_layer=params["number_of_layer"],

module__number_of_hidden_layer_neurons=params["number_of_hidden_layer_neuro
ns"],
            module__activation_function = params["activation_function"],
            module__dropout_probability = params["dropout_probability"],
            module__use_batchnorm = params["use_batchnorm"],

            criterion=torch.nn.MSELoss,

            batch_size = params["batch_size"],
            max_epochs = params["max_epochs"],

            optimizer=torch.optim.Adam,
            optimizer__lr=params["lr"],
            optimizer__weight_decay=params["weight_decay"],

            device= 'cuda' if torch.cuda.is_available() else "cpu"
        )


        model.fit(X_train, y_train)

        y_pred_train = model.predict(X_train)
        y_pred_test = model.predict(X_test)

        mse_train = mean_squared_error(y_train, y_pred_train)
        mse_test = mean_squared_error(y_test, y_pred_test)
    #     print("MSE train: ", mse_train)
    #     print("MSE test: ", mse_test)

        row["mse_train"] = mse_train
        row["mse_test"] = mse_test

        search_space_result_df = search_space_result_df.append(row,
ignore_index=True)
    except Exception:
        pass

    if i%500:
        search_space_result_df.to_csv("search_space_result_NN.csv")
        search_space_result_df.to_excel("search_space_result_NN.xlsx")

from sklearn.metrics import mean_squared_error

X_train  = X_train.astype(np.float32)
y_train = y_train.astype(np.float32)
X_test = X_test.astype(np.float32)
y_test = y_test.astype(np.float32)


model.fit(X_train, y_train)
```

```
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

mse_train = mean_squared_error(y_train, y_pred_train)
mse_test = mean_squared_error(y_test, y_pred_test)

print("MSE train: ", mse_train)
print("MSE test: ", mse_test)
```

Załącznik nr 4 – kod źródłowy skryptu 2 do tworzenia modeli w ABAQUS

```
from part import *
from material import *
from section import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from optimization import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *

import math

#Models quantity
Mq = 315

class VIGmodel:
    def dimensions(self, A_dimension, B_dimension, thickness_glass,
thickness_vacuum, sealing_width):
        self.A  = A_dimension
        self.B  = B_dimension
        self.tg = thickness_glass
        self.tv = thickness_vacuum
        self.ws = sealing_width

    def pillarsgeom(self, x_quantity, y_quantity, diameter,
first_x_location, first_y_location):
        self.nx = x_quantity
        self.ny = y_quantity
        self.dp = diameter
        self.xp = first_x_location
        self.yp = first_y_location

    def bandsgeom(self, Length, length, heigth, width, thickness,
foundation, Thickness):
        self.Lb = Length
        self.lb = length
        self.hb = heigth
        self.wb = width
        self.tb = thickness
        self.Ez = foundation
        self.Tb = Thickness

    def glassprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_g = density
        self.E_g = modulus_elasticity
        self.v_g = friction_Poisson

    def pillarsprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_p = density
        self.E_p = modulus_elasticity
        self.v_p = friction_Poisson

    def sealingprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_s = density
        self.E_s = modulus_elasticity
        self.v_s = friction_Poisson
```

```
    def bandsprop(self, density, modulus_elasticity, friction_Poisson):
        self.d_b = density
        self.E_b = modulus_elasticity
        self.v_b = friction_Poisson

    def glassmesh(self, deviationFactor, minSizeFactor, size,
thickness_divide, finer_width_x, finer_width_y, finer_deviationFactor,
finer_minSizeFactor, finer_size):
        self.dF_g    = deviationFactor
        self.mSF_g   = minSizeFactor
        self.sz_g    = size
        self.tdv_g   = thickness_divide
        self.fw_x    = finer_width_x
        self.fw_y    = finer_width_y
        self.f_dF_g  = finer_deviationFactor
        self.f_mSF_g = finer_minSizeFactor
        self.f_sz_g  = finer_size

    def pillarsmesh(self, deviationFactor, minSizeFactor, size):
        self.dF_p  = deviationFactor
        self.mSF_p = minSizeFactor
        self.sz_p  = size

    def sealingmesh(self, deviationFactor, minSizeFactor, size,
thickness_divide):
        self.dF_s  = deviationFactor
        self.mSF_s = minSizeFactor
        self.sz_s  = size
        self.tdv_s = thickness_divide

    def bandsmesh(self, deviationFactor, minSizeFactor, size):
        self.dF_b  = deviationFactor
        self.mSF_b = minSizeFactor
        self.sz_b  = size

    def create(self, model_num, eigenv_num, load_q):
        self.Mn = model_num
        self.En = eigenv_num
        self.Q  = load_q

        #Delete and create model
        mdb.Model(modelType=STANDARD_EXPLICIT, name='VIG' + str(self.Mn))

        #Create Glass1
        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=2.0)
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(0.0, 0.0),
point2=(self.A, self.B))
        mdb.models['VIG' + str(self.Mn)].Part(dimensionality=THREE_D,
name='Glass_1', type=DEFORMABLE_BODY)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].BaseSolidExtrude(depth=self.tg,
sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']

        #Create Glass2
        mdb.models['VIG' + str(self.Mn)].Part(name='Glass_2',
objectToCopy=mdb.models['VIG' + str(self.Mn)].parts['Glass_1'])

        #Create Sealing
        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=2.0)
```

```
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(0.0, 0.0),
point2=(self.A, self.B))
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(self.ws, self.ws),
point2=(self.A-self.ws, self.B-self.ws))
        mdb.models['VIG' + str(self.Mn)].Part(dimensionality=THREE_D,
name='Sealing', type=DEFORMABLE_BODY)
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].BaseSolidExtrude(depth=self.tv,
sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']

        #Change elements' names
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.DatumCsysByDefault(CARTESIAN)
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Glass_1-1',
part=mdb.models['VIG' + str(self.Mn)].parts['Glass_1'])
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Glass_2-1',
part=mdb.models['VIG' + str(self.Mn)].parts['Glass_2'])
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Sealing-1',
part=mdb.models['VIG' + str(self.Mn)].parts['Sealing'])

        #Create pillars
        Pillars = list()
        Pillars_cuts = list()
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=2.0)
                mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].CircleByCenterPerimeter(center=(self.
xp + (self.A-2*self.xp)/(self.nx-1)*i, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j), point1=(self.xp + (self.A-2*self.xp)/(self.nx-
1)*i + 0.5*self.dp, self.yp + (self.B-2*self.yp)/(self.ny-1)*j))
                mdb.models['VIG' +
str(self.Mn)].Part(dimensionality=THREE_D, name='Pillar_' + str(i) + '_' +
str(j), type=DEFORMABLE_BODY)
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].BaseSolidExtrude(depth=self.tv, sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'])
                del mdb.models['VIG' +
str(self.Mn)].sketches['__profile__']
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Pillar-' + str(i) +
'_' + str(j), part=mdb.models['VIG' + str(self.Mn)].parts['Pillar_' +
str(i) + '_' + str(j)])

                Pillars.append('Pillar-' + str(i) + '_' + str(j))
                Pillars_cuts.append(mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Pillar-' + str(i) + '_' + str(j)])

        #Pillars - second add and translate
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Pillar-' + str(i) +
'_' + str(j), part=mdb.models['VIG' + str(self.Mn)].parts['Pillar_' +
str(i) + '_' + str(j)])

        #Create bands
```

```
        for i in range (0,4):
            mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=1.0)
            mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(0.0, 0.0),
point2=(self.wb, self.Lb))
            mdb.models['VIG' + str(self.Mn)].Part(dimensionality=THREE_D,
name='Band_' + str(i), type=DEFORMABLE_BODY)
            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].BaseShell(sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'])
            del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']

            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].engineeringFeatures.SpringDashpotToGround(dashpotBehavior=OFF,
dashpotCoefficient=0.0, dof=2, name='Springs/Dashpots-1', orientation=None,
            region=Region(vertices=mdb.models['VIG' +
str(self.Mn)].parts['Band_0'].vertices.getSequenceFromMask(mask=('[#c ]',
), )), springBehavior=ON,
springStiffness=0.5*((self.Ez*self.wb*self.tb)/(((0.5*(self.lb -
self.B))**2.0 + (0.5*self.hb)**2.0)**0.5)))

        for i in range (4,8):
            mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(name='__profile__', sheetSize=1.0)
            mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(0.0, 0.0),
point2=(self.wb, self.Lb))
            mdb.models['VIG' + str(self.Mn)].Part(dimensionality=THREE_D,
name='Band_' + str(i), type=DEFORMABLE_BODY)
            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].BaseShell(sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'])
            del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']

            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].engineeringFeatures.SpringDashpotToGround(dashpotBehavior=OFF,
dashpotCoefficient=0.0, dof=2, name='Springs/Dashpots-1', orientation=None,
            region=Region(vertices=mdb.models['VIG' +
str(self.Mn)].parts['Band_0'].vertices.getSequenceFromMask(mask=('[#c ]',
), )), springBehavior=ON,
springStiffness=0.5*((self.Ez*self.wb*self.tb)/(((0.5*(self.lb -
self.A))**2.0 + (0.5*self.hb)**2.0)**0.5)))

        for i in range (8):
            mdb.models['VIG' +
str(self.Mn)].rootAssembly.Instance(dependent=ON, name='Band-' +
str(i),part=mdb.models['VIG' + str(self.Mn)].parts['Band_' + str(i)])


        #Materials
        mdb.models['VIG' + str(self.Mn)].Material(name='Glass')
        mdb.models['VIG' +
str(self.Mn)].materials['Glass'].Density(table=((self.d_g, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Glass'].Elastic(table=((self.E_g, self.v_g), ))

        mdb.models['VIG' + str(self.Mn)].Material(name='Pillar')
        mdb.models['VIG' +
str(self.Mn)].materials['Pillar'].Density(table=((self.d_p, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Pillar'].Elastic(table=((self.E_p, self.v_p), ))

        mdb.models['VIG' + str(self.Mn)].Material(name='Sealing')
```

```
        mdb.models['VIG' +
str(self.Mn)].materials['Sealing'].Density(table=((self.d_s, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Sealing'].Elastic(table=((self.E_s, self.v_s), ))

        mdb.models['VIG' + str(self.Mn)].Material(name='Band')
        mdb.models['VIG' +
str(self.Mn)].materials['Band'].Density(table=((self.d_b, ), ))
        mdb.models['VIG' +
str(self.Mn)].materials['Band'].Elastic(table=((self.E_b, self.v_b), ))


        #Sections
        mdb.models['VIG' +
str(self.Mn)].HomogeneousSolidSection(material='Glass', name='Glass',
thickness=None)
        mdb.models['VIG' +
str(self.Mn)].HomogeneousSolidSection(material='Pillar', name='Pillar',
thickness=None)
        mdb.models['VIG' +
str(self.Mn)].HomogeneousSolidSection(material='Sealing', name='Sealing',
thickness=None)
        mdb.models['VIG' +
str(self.Mn)].HomogeneousShellSection(idealization=NO_IDEALIZATION,
integrationRule=SIMPSON, material='Band', name='Band',
nodalThicknessField='', numIntPts=5, poissonDefinition=DEFAULT,
            preIntegrate=OFF, temperature=GRADIENT, thickness=self.Tb,
thicknessField='', thicknessModulus=None, thicknessType=UNIFORM,
useDensity=OFF)

        #Sections assignment
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].Set(cells=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].cells.getSequenceFromMask(('[#1 ]', ), ),
name='Set-1')
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].sets['Set-1'], sectionName='Glass',
thicknessAssignment=FROM_SECTION)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].Set(cells=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].cells.getSequenceFromMask(('[#1 ]', ), ),
name='Set-1')
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].sets['Set-1'], sectionName='Glass',
thicknessAssignment=FROM_SECTION)
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].Set(cells=mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].cells.getSequenceFromMask(('[#1 ]', ), ),
name='Set-1')
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].sets['Set-1'], sectionName='Sealing',
thicknessAssignment=FROM_SECTION)

        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].Set(cells=mdb.models['VIG' + str(self.Mn)].parts['Pillar_' +
```

```
str(i) + '_' + str(j)].cells.getSequenceFromMask(('[#1 ]', ), ), name='Set-
1')
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].SectionAssignment(offset=0.0, offsetField='',
offsetType=MIDDLE_SURFACE,
                    region=mdb.models['VIG' + str(self.Mn)].parts['Pillar_'
+ str(i) + '_' + str(j)].sets['Set-1'], sectionName='Pillar',
thicknessAssignment=FROM_SECTION)

        for i in range (8):
            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].Set(faces=mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].faces.getSequenceFromMask(('[#1 ]', ),), name='Set-1')
            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].SectionAssignment(offset=0.0, offsetField='',
offsetType=MIDDLE_SURFACE, region=mdb.models['VIG' +
str(self.Mn)].parts['Band_' + str(i)].sets['Set-1'], sectionName='Band',
thicknessAssignment=FROM_SECTION)


        #Translations
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Glass_2-1', ),
vector=(0.0, 0.0, self.tg+self.tv))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Sealing-1', ),
vector=(0.0, 0.0, self.tg))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=Pillars, vector=(0.0,
0.0, self.tg))

        mdb.models['VIG' + str(self.Mn)].rootAssembly.rotate(angle=
math.atan(self.hb/2/(0.5*(self.lb-self.B)))*180/math.pi - 180,
axisDirection=(-1.0, 0.0, 0.0), axisPoint=(0.0, 0.0, 0.0),
instanceList=('Band-0', 'Band-1'))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.rotate(angle=-
math.atan(self.hb/2/(0.5*(self.lb-self.B)))*180/math.pi, axisDirection=(-
1.0, 0.0, 0.0), axisPoint=(0.0, 0.0, 0.0), instanceList=('Band-2', 'Band-
3'))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.rotate(angle=
math.atan(self.hb/2/(0.5*(self.lb-self.A)))*180/math.pi - 180,
axisDirection=(-1.0, 0.0, 0.0), axisPoint=(0.0, 0.0, 0.0),
instanceList=('Band-4', 'Band-5'))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.rotate(angle=-
math.atan(self.hb/2/(0.5*(self.lb-self.A)))*180/math.pi, axisDirection=(-
1.0, 0.0, 0.0), axisPoint=(0.0, 0.0, 0.0), instanceList=('Band-6', 'Band-
7'))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Band-2', 'Band-3',
'Band-6', 'Band-7'), vector=(0.0, self.B, 0.0))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Band-1', 'Band-3'),
vector=(self.A - self.wb, 0.0, 0.0))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.rotate(angle=-90.0,
axisDirection=(0.0, 0.0, 1.0), axisPoint=(0.5*self.wb, 0.5*self.wb, 0.0),
instanceList=('Band-4', 'Band-5', 'Band-6', 'Band-7'))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Band-6', 'Band-7'),
vector=(self.A - self.B, 0.0, 0.0))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.translate(instanceList=('Band-5', 'Band-7'),
vector=(0.0, self.B - self.wb, 0.0))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.rotate(angle=-180.0,
axisDirection=(-1.0, 0.0, 0.0), axisPoint=(0.5*self.A, 0.5*self.B,
```

```
0.5*(2*self.tg+self.tv)), instanceList=('Band-4', 'Band-5', 'Band-6',
'Band-7'))


        #Surfaces
        mdb.models['VIG' + str(self.Mn)].rootAssembly.regenerate()
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass1_int',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, self.tg), )))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass1_ext',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, 0), )))
        mdb.models['VIG' + str(self.Mn)].rootAssembly.regenerate()
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass2_int',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, self.tg + self.tv), )))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Glass2_ext',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(((0.5*self.A, 0.5*self.B, 2*self.tg + self.tv), )))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Sealing_int_g1',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Sealing-
1'].faces.findAt(((0.5*self.ws, 0.5*self.ws, self.tg), )))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Sealing_int_g2',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Sealing-
1'].faces.findAt(((0.5*self.ws, 0.5*self.ws, self.tg + self.tv), )))


        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Pillar_' + str(i) + '_' + str(j) +
'_int_g1', side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Pillar-' + str(i) + '_' +
str(j)].faces.getSequenceFromMask(('[#4 ]', ), ))
                mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='Pillar_' + str(i) + '_' + str(j) +
'_int_g2', side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Pillar-' + str(i) + '_' +
str(j)].faces.getSequenceFromMask(('[#2 ]', ), ))



        #Ties
        mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass1_int'], name='Glass1_sealing',
positionToleranceMethod=COMPUTED, slave=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Sealing_int_g1'], thickness=ON,
tieRotations=ON)
```

```
        mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass2_int'], name='Glass2_sealing',
positionToleranceMethod=COMPUTED, slave=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Sealing_int_g2'], thickness=ON,
tieRotations=ON)

        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass1_int'], name='Glass1_pillar_' +
str(i) + '_' + str(j), positionToleranceMethod=COMPUTED,
slave=mdb.models['VIG' + str(self.Mn)].rootAssembly.surfaces['Pillar_' +
str(i) + '_' + str(j) + '_int_g1'], thickness=ON, tieRotations=ON)
                mdb.models['VIG' + str(self.Mn)].Tie(adjust=ON,
master=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass2_int'], name='Glass2_pillar_' +
str(i) + '_' + str(j), positionToleranceMethod=COMPUTED,
slave=mdb.models['VIG' + str(self.Mn)].rootAssembly.surfaces['Pillar_' +
str(i) + '_' + str(j) + '_int_g2'], thickness=ON, tieRotations=ON)

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band0',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
0'].edges.findAt(coordinates=tuple([(0.5*self.wb, 0.0, 0.0)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass1_0',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(coordinates=tuple([(0.5*self.wb, 0.0, 0.5*self.tg)])))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band2',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
1'].edges.findAt(coordinates=tuple([(self.A - 0.5*self.wb, 0.0, 0.0)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass1_2',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(coordinates=tuple([(self.A - 0.5*self.wb, 0.0,
0.5*self.tg)])))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band1',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
2'].edges.findAt(coordinates=tuple([(0.5*self.wb, self.B, 0.0)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass1_1',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(coordinates=tuple([(0.5*self.wb, self.B, 0.5*self.tg)])))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band3',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
3'].edges.findAt(coordinates=tuple([(self.A - 0.5*self.wb, self.B, 0.0)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass1_3',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_1-
1'].faces.findAt(coordinates=tuple([(self.A - 0.5*self.wb, self.B,
0.5*self.tg)])))
```

```
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band4',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
4'].edges.findAt(coordinates=tuple([(0.0, self.B - 0.5*self.wb,
2*self.tg+self.tv)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass2_4',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(coordinates=tuple([(0.0, self.B - 0.5*self.wb,
1.5*self.tg+self.tv)])))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band5',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
5'].edges.findAt(coordinates=tuple([(0.0, 0.5*self.wb,
2*self.tg+self.tv)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass2_5',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(coordinates=tuple([(0.0, 0.5*self.wb,
1.5*self.tg+self.tv)])))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band6',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
6'].edges.findAt(coordinates=tuple([(self.A, self.B - 0.5*self.wb,
2*self.tg+self.tv)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass2_6',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(coordinates=tuple([(self.A, self.B - 0.5*self.wb,
1.5*self.tg+self.tv)])))

        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='m_band7',
side1Edges=mdb.models['VIG' + str(self.Mn)].rootAssembly.instances['Band-
7'].edges.findAt(coordinates=tuple([(self.A, 0.5*self.wb,
2*self.tg+self.tv)])))
        mdb.models['VIG' +
str(self.Mn)].rootAssembly.Surface(name='s_glass2_7',
side1Faces=mdb.models['VIG' +
str(self.Mn)].rootAssembly.instances['Glass_2-
1'].faces.findAt(coordinates=tuple([(self.A, 0.5*self.wb,
1.5*self.tg+self.tv)])))

        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band0_Glass1',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band0'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass1_0'])
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band1_Glass1',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band1'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass1_1'])
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band2_Glass1',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band2'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass1_2'])
```

```
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band3_Glass1',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band3'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass1_3'])
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band4_Glass2',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band4'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass2_4'])
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band5_Glass2',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band5'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass2_5'])
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band6_Glass2',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band6'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass2_6'])
        mdb.models['VIG' +
str(self.Mn)].ShellSolidCoupling(name='Band7_Glass2',
positionToleranceMethod=COMPUTED, shellEdge=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['m_band7'], solidFace=mdb.models['VIG'
+ str(self.Mn)].rootAssembly.surfaces['s_glass2_7'])



        #Mesh
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].seedPart(deviationFactor=self.dF_g,
minSizeFactor=self.mSF_g, size=self.sz_g)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].seedEdgeByNumber(constraint=FINER,
edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].edges.getSequenceFromMask(('[#20 ]', ),),
number=self.tdv_g)

        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(gridSpacing=0.005,
name='__profile__',sheetSize=0.212,
            transform=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].MakeSketchTransform(sketchPlane=mdb.models['
VIG' + str(self.Mn)].parts['Glass_1'].faces[4], sketchPlaneSide=SIDE1,
            sketchUpEdge=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].edges[10], sketchOrientation=BOTTOM,
origin=(0.5*self.A, 0.5*self.B, self.tg)))
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].sketchOptions.setValues(decimalPlaces
=3)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].projectReferencesOntoSketch(filter=COPLANAR_
EDGES, sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        finer_edges_g1 = []
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x - 0.5*self.A, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j - 0.5*self.fw_y - 0.5*self.B), point2=(self.xp +
(self.A-2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x - 0.5*self.A, self.yp +
(self.B-2*self.yp)/(self.ny-1)*j + 0.5*self.fw_y - 0.5*self.B))
```

```
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j -
0.5*self.fw_y, self.tg))
                finer_edges_g1.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j +
0.5*self.fw_y, self.tg))
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].PartitionFaceBySketch(faces=mdb.models['VIG'
+ str(self.Mn)].parts['Glass_1'].faces.getSequenceFromMask(('[#10 ]', ),),
            sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'], sketchOrientation=BOTTOM,
sketchUpEdge=mdb.models['VIG' + str(self.Mn)].parts['Glass_1'].edges[10])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].seedEdgeBySize(constraint=FINER,
deviationFactor=self.f_dF_g, edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_1'].edges.findAt(coordinates=tuple(finer_edges_g
1)), minSizeFactor=self.f_mSF_g, size=self.f_sz_g)
        mdb.models['VIG' + str(self.Mn)].parts['Glass_1'].generateMesh()


        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].seedPart(deviationFactor=self.dF_g,
minSizeFactor=self.mSF_g, size=self.sz_g)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].seedEdgeByNumber(constraint=FINER,
edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].edges.getSequenceFromMask(('[#20 ]', ),),
number=self.tdv_g)

        mdb.models['VIG' +
str(self.Mn)].ConstrainedSketch(gridSpacing=0.005,
name='__profile__',sheetSize=0.212,
            transform=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].MakeSketchTransform(sketchPlane=mdb.models['
VIG' + str(self.Mn)].parts['Glass_2'].faces[4], sketchPlaneSide=SIDE1,
            sketchUpEdge=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].edges[10], sketchOrientation=BOTTOM,
origin=(0.5*self.A, 0.5*self.B, self.tg)))
        mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].sketchOptions.setValues(decimalPlaces
=3)
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].projectReferencesOntoSketch(filter=COPLANAR_
EDGES, sketch=mdb.models['VIG' + str(self.Mn)].sketches['__profile__'])
        finer_edges_g2 = []
        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'].rectangle(point1=(self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x - 0.5*self.A, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j - 0.5*self.fw_y - 0.5*self.B), point2=(self.xp +
(self.A-2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x - 0.5*self.A, self.yp +
(self.B-2*self.yp)/(self.ny-1)*j + 0.5*self.fw_y - 0.5*self.B))
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i - 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
```

```
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i + 0.5*self.fw_x, self.yp + (self.B-
2*self.yp)/(self.ny-1)*j, self.tg))
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j -
0.5*self.fw_y, self.tg))
                finer_edges_g2.append((self.xp + (self.A-
2*self.xp)/(self.nx-1)*i, self.yp + (self.B-2*self.yp)/(self.ny-1)*j +
0.5*self.fw_y, self.tg))
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].PartitionFaceBySketch(faces=mdb.models['VIG'
+ str(self.Mn)].parts['Glass_2'].faces.getSequenceFromMask(('[#10 ]', ),),
            sketch=mdb.models['VIG' +
str(self.Mn)].sketches['__profile__'], sketchOrientation=BOTTOM,
sketchUpEdge=mdb.models['VIG' + str(self.Mn)].parts['Glass_2'].edges[10])
        del mdb.models['VIG' + str(self.Mn)].sketches['__profile__']
        mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].seedEdgeBySize(constraint=FINER,
deviationFactor=self.f_dF_g, edges=mdb.models['VIG' +
str(self.Mn)].parts['Glass_2'].edges.findAt(coordinates=tuple(finer_edges_g
2)), minSizeFactor=self.f_mSF_g, size=self.f_sz_g)
        mdb.models['VIG' + str(self.Mn)].parts['Glass_2'].generateMesh()


        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].seedPart(deviationFactor=self.dF_s,
minSizeFactor=self.mSF_s, size=self.sz_s)
        mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].seedEdgeByNumber(constraint=FINER,
edges=mdb.models['VIG' +
str(self.Mn)].parts['Sealing'].edges.getSequenceFromMask(('[#20 ]', ),),
number=self.tdv_s)
        mdb.models['VIG' + str(self.Mn)].parts['Sealing'].generateMesh()


        for i in range (self.nx):
            for j in range (self.ny):
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].seedPart(deviationFactor=self.dF_p, minSizeFactor=self.mSF_p,
size=self.sz_p)
                mdb.models['VIG' + str(self.Mn)].parts['Pillar_' + str(i) +
'_' + str(j)].generateMesh()


        for i in range (8):
            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].seedPart(deviationFactor=self.dF_b, minSizeFactor=self.mSF_b,
size=self.sz_b)
            mdb.models['VIG' + str(self.Mn)].parts['Band_' +
str(i)].generateMesh()


        #Step - dynamics
        mdb.models['VIG' + str(self.Mn)].FrequencyStep(name='Step-1',
numEigen= self.En , previous='Initial')

        #Step - statics
        mdb.models['VIG' + str(self.Mn)].StaticStep(description='Static',
name='Step-2', previous='Step-1')
        mdb.models['VIG' + str(self.Mn)].steps['Step-
2'].setValues(initialInc=0.01)
        mdb.models['VIG' + str(self.Mn)].Pressure(amplitude=UNSET,
createStepName='Step-2', distributionType=UNIFORM, field='',
```

```
magnitude=self.Q, name='Vacuum1', region=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass1_ext'])
        mdb.models['VIG' + str(self.Mn)].Pressure(amplitude=UNSET,
createStepName='Step-2', distributionType=UNIFORM, field='',
magnitude=self.Q, name='Vacuum2', region=mdb.models['VIG' +
str(self.Mn)].rootAssembly.surfaces['Glass2_ext'])

        #Jobs
        mdb.Job(atTime=None, contactPrint=OFF, description='',
echoPrint=OFF, explicitPrecision=SINGLE, getMemoryFromAnalysis=True,
historyPrint=OFF,
            memory=90, memoryUnits=PERCENTAGE, model='VIG' + str(self.Mn),
modelPrint=OFF, name='Job-' + str(self.Mn), nodalOutputPrecision=SINGLE,
queue=None, resultsFormat=ODB,
            scratch='', type=ANALYSIS, userSubroutine='', waitHours=0,
waitMinutes=0)


        #Model regeneration
        mdb.models['VIG' + str(self.Mn)].rootAssembly.regenerate()

    def jobSubmit(self):
        mdb.jobs['Job-' + str(self.Mn)].submit(consistencyChecking=OFF)
        mdb.jobs['Job-' + str(self.Mn)].waitForCompletion()

    def results_save(self, model_num):
        self.Mn = model_num

        odb = openOdb(path='/home/grid/users/plgdamkoz/Job-' + str(self.Mn)
+ '.odb')
        frames=odb.steps['Step-1'].frames

        TxtVal=open("__1_EIGENv.txt", 'a')
        TxtFreq=open("__1_EIGENf.txt", 'a')

        TxtVal.write('Model_' + str(self.Mn) + ';')
        TxtFreq.write('Model_' + str(self.Mn) + ';')

        for frame in frames:
            f=frame.description
            if len(f[28:48])>1:
                TxtVal.write(str(float(f[24:37])) + ';')
                TxtFreq.write(str(float(f[44:57])) + ';')

        TxtVal.write('\n')
        TxtFreq.write('\n')

        TxtVal.close()
        TxtFreq.close()

        TxtDispl=open("__2_DISPL.txt", 'a')
        TxtTens=open("__2_TENS.txt", 'a')

        lastframe=odb.steps['Step-2'].frames[-1]
        displacements = lastframe.fieldOutputs['U'].values
        stresses = lastframe.fieldOutputs['S'].values

        displacementsL = []
        for v in displacements:
            displacementsL.append(v.data[2])

        TxtDispl.write('Model_' + str(self.Mn) + ';')
        TxtDispl.write(str(min(displacementsL)) + ';')
        TxtDispl.write(str(max(displacementsL)) + ';')
```

```
        stressesL = []
        for v in stresses:
            stressesL.append(v.mises)

        TxtTens.write('Model_' + str(self.Mn) + ';')
        TxtTens.write(str(min(stressesL)) + ';')
        TxtTens.write(str(max(stressesL)) + ';')

        TxtDispl.write('\n')
        TxtTens.write('\n')

        TxtDispl.close()
        TxtTens.close()


#CREATING DATA
VIG_P = [[[0 for x in range(5)], [0 for x in range(5)], [0 for x in
range(7)], [0 for x in range(3)], [0 for x in range(3)], [0 for x in
range(3)], [0 for x in range(3)]] for x in range(Mq)]

for i in range(Mq):
    #DIMENSIONS
    #A
    if i < 105:
        VIG_P[i][0][0] = 0.3
    elif i < 189:
        VIG_P[i][0][0] = 0.6
    elif i < 252:
        VIG_P[i][0][0] = 0.9
    elif i < 294:
        VIG_P[i][0][0] = 1.2
    else:
        VIG_P[i][0][0] = 1.5

    #B
    if i < 21:
        VIG_P[i][0][1] = 0.3
    elif i < 42:
        VIG_P[i][0][1] = 0.6
    elif i < 63:
        VIG_P[i][0][1] = 0.9
    elif i < 84:
        VIG_P[i][0][1] = 1.2
    elif i < 105:
        VIG_P[i][0][1] = 1.5
    elif i < 126:
        VIG_P[i][0][1] = 0.6
    elif i < 147:
        VIG_P[i][0][1] = 0.9
    elif i < 168:
        VIG_P[i][0][1] = 1.2
    elif i < 189:
        VIG_P[i][0][1] = 1.5
    elif i < 210:
        VIG_P[i][0][1] = 0.9
    elif i < 231:
        VIG_P[i][0][1] = 1.2
    elif i < 252:
        VIG_P[i][0][1] = 1.5
    elif i < 273:
        VIG_P[i][0][1] = 1.2
    elif i < 294:
        VIG_P[i][0][1] = 1.5
```

```
    else:
        VIG_P[i][0][1] = 1.5

    #glass thickness
    if (i)%21 == 0:
        VIG_P[i][0][2] = 0.004
    else:
        if (i)%7 == 0:
            VIG_P[i][0][2] = VIG_P[i-1][0][2] + 0.001
        else:
            VIG_P[i][0][2] = VIG_P[i-1][0][2]

    VIG_P[i][0][3] = 0.0003

    #sealing width
    VIG_P[i][0][4] = 0.009

    #PILLARSGEOM
    #quantity
    if VIG_P[i][0][0] == 0.3:
        VIG_P[i][1][0] = 5
    elif VIG_P[i][0][0] == 0.6:
        VIG_P[i][1][0] = 10
    elif VIG_P[i][0][0] == 0.9:
        VIG_P[i][1][0] = 15
    elif VIG_P[i][0][0] == 1.2:
        VIG_P[i][1][0] = 21
    elif VIG_P[i][0][0] == 1.5:
        VIG_P[i][1][0] = 26

    if VIG_P[i][0][1] == 0.3:
        VIG_P[i][1][1] = 5
    elif VIG_P[i][0][1] == 0.6:
        VIG_P[i][1][1] = 10
    elif VIG_P[i][0][1] == 0.9:
        VIG_P[i][1][1] = 15
    elif VIG_P[i][0][1] == 1.2:
        VIG_P[i][1][1] = 21
    elif VIG_P[i][0][1] == 1.5:
        VIG_P[i][1][1] = 26

    VIG_P[i][1][2] = 0.0006

    if VIG_P[i][1][0] == 5:
        VIG_P[i][1][3] = 0.04
    elif VIG_P[i][1][0] == 10:
        VIG_P[i][1][3] = 0.0525
    elif VIG_P[i][1][0] == 15:
        VIG_P[i][1][3] = 0.065
    elif VIG_P[i][1][0] == 21:
        VIG_P[i][1][3] = 0.050
    elif VIG_P[i][1][0] == 26:
        VIG_P[i][1][3] = 0.0625

    if VIG_P[i][1][1] == 5:
        VIG_P[i][1][4] = 0.04
    elif VIG_P[i][1][1] == 10:
        VIG_P[i][1][4] = 0.0525
    elif VIG_P[i][1][1] == 15:
        VIG_P[i][1][4] = 0.065
    elif VIG_P[i][1][1] == 21:
        VIG_P[i][1][4] = 0.050
    elif VIG_P[i][1][1] == 26:
        VIG_P[i][1][4] = 0.0625
```

```
    #BANDSGEOM
    VIG_P[i][2][0] = 0.002
    VIG_P[i][2][1] = 2.0
    VIG_P[i][2][2] = 1.0
    VIG_P[i][2][3] = 0.025
    VIG_P[i][2][4] = 0.00075
    VIG_P[i][2][5] = 4.0*(10**9)
    VIG_P[i][2][6] = 0.01

    #GLASSPROP
    VIG_P[i][3][0] = 2500.0
    VIG_P[i][3][1] = 72.0*(10**9)
    VIG_P[i][3][2] = 0.22

    #PILLARSPROP
    VIG_P[i][4][0] = 7850.0
    VIG_P[i][4][2] = 0.31

    if (i)%7 == 0:
        VIG_P[i][4][1] = 160.0*(10**9)
    else:
        VIG_P[i][4][1] = VIG_P[i-1][4][1] + 10.0*(10**9)

    #SEALINGPROP
    VIG_P[i][5][0] = 7850.0
    VIG_P[i][5][1] = 210.0*(10**9)
    VIG_P[i][5][2] = 0.31

    #BANDSPROP
    VIG_P[i][6][0] = 10.0
    VIG_P[i][6][1] = 1000.0*(10**9)
    VIG_P[i][6][2] = 0.33

#RUN CODE
VIGlist = list()

for i in range(Mq):
       VIGlist.append(VIGmodel())

for i in range(Mq):
    VIGlist[i].dimensions (VIG_P[i][0][0], VIG_P[i][0][1], VIG_P[i][0][2],
VIG_P[i][0][3], VIG_P[i][0][4])
    VIGlist[i].pillarsgeom(VIG_P[i][1][0], VIG_P[i][1][1], VIG_P[i][1][2],
VIG_P[i][1][3], VIG_P[i][1][4])
    VIGlist[i].bandsgeom  (VIG_P[i][2][0], VIG_P[i][2][1], VIG_P[i][2][2],
VIG_P[i][2][3], VIG_P[i][2][4], VIG_P[i][2][5], VIG_P[i][2][6])

    VIGlist[i].glassprop  (VIG_P[i][3][0], VIG_P[i][3][1], VIG_P[i][3][2])
    VIGlist[i].pillarsprop(VIG_P[i][4][0], VIG_P[i][4][1], VIG_P[i][4][2])
    VIGlist[i].sealingprop(VIG_P[i][5][0], VIG_P[i][5][1], VIG_P[i][5][2])
    VIGlist[i].bandsprop  (VIG_P[i][6][0], VIG_P[i][6][1], VIG_P[i][6][2])

    VIGlist[i].glassmesh(0.95, 0.95, 0.0101, 2, 0.001, 0.001, 0.95, 0.95,
0.001)
    VIGlist[i].pillarsmesh(0.95, 0.95, 0.0001)
    VIGlist[i].sealingmesh(0.95, 0.95, 0.005, 2)
    VIGlist[i].bandsmesh(0.95, 0.95, 0.01)

    VIGlist[i].create(i, 30, 100000)

for i in range(Mq):
    VIGlist[i].jobSubmit()
    VIGlist[i].results_save(i)
```

Streszczenie

Przedmiotem niniejszej pracy są płyty typu VIG (Vacuum Insulated Glass). Celem pracy było znalezienie modelu numerycznego, który pozwoli na efektywne i dokładne odwzorowanie numeryczne rzeczywistej płyty przy uwzględnieniu odpowiednich warunków podporowych. Dodatkowym celem było przeprowadzenie analizy wpływu różnych parametrów geometrycznych i mechanicznych na wartości częstotliwości drgań własnych oraz opis i dobór metod sztucznej inteligencji i zbiorów danych, w celu określenia modułu sprężystości podłużnej pilastrów znajdujących się wewnątrz analizowanych szyb próżniowych, przy użyciu stworzonych modeli predykcyjnych.

Pracę podzielono na cztery części. W pierwszej z nich przedstawiono zastosowane metody analizy, w drugiej przeanalizowano dynamiczne zachowanie płyt typu VIG, a w trzeciej zaprezentowane zostały zastosowane metody sztucznej inteligencji. W czwartej części, metody oraz wyniki uzyskane z trzech poprzednich zostały wykorzystane w odniesieniu do badań laboratoryjnych.

Summary

The subject of this thesis is VIG (Vacuum Insulated Glass) plate. The aim of this study was to find a numerical model in order to create an effective and precise numerical imitation of the real panel, considering the appropriate support conditions. Furthermore, the aim was also to analyse the influence of various geometrical and mechanical parameters on the natural frequencies values as well as the description and selection of artificial intelligence methods and data sets, in order to determine the value of Young's modulus of pillars located between glass panes, using the created predictive models.

The paper was divided into four parts. The first one presents the used methods of analysis, the second one analyses of the dynamic behaviour of VIG plates, and the third one presents the applied methods of artificial intelligence. In the fourth part, the methods and results from the first three ones were used in relation to laboratory tests.