Transmisja danych dźwiękiem w JavaScript od podstaw

Część 1: Dyskretna Transformata Fouriera

Przez lata JavaScript stał się językiem o ogromnym potencjale. W nowoczesnych przeglądarkach znajdziemy API do wielu różnych zastosowań. Możemy np. rysować grafikę 2D/3D przy użyciu WebGL czy też użyć WebWorkerów dla zwiększenia wydajności bardziej złożonych obliczeń na procesorach wielordzeniowych. Możemy także obsługiwać sprzęt, taki jak mikrofon czy głośniki poprzez Web Audio API. Lista wszystkich interfejsów oferowanych przez przeglądarki z biegiem lat staje się coraz dłuższa. Sprawia to, że JS staje się coraz bardziej popularny, a w przypadku urządzeń mobilnych aplikacje często konkurują z rozwiązaniami natywnymi.

Celem tego artykułu jest wykorzystanie Web Audio API do powrotu do starych czasów. Czy pamiętamy jeszcze dźwięk modemu inicjującego wdzwaniane połączenie? Był on dość hałaśliwy i trwał kilka sekund. Po poprawnej inicjalizacji dane były transmitowane i odbierane poprzez zwykłe połączenie telefoniczne. W zasadzie można było „posłuchać" tych danych, podnosząc słuchawkę telefonu podłączonego do tej samej linii. Dlaczego dźwięk? Dlatego że w tamtych czasach wielu ludzi posiadało już linię telefoniczną. Użycie modemu oraz istniejącej infrastruktury było po prostu tańsze i prostsze.

Nie ma jednak róży bez kolców. Systemy telefoniczne w tamtych czasach były projektowane tylko do przesyłania ludzkiej mowy w bardzo ograniczonym zakresie. Cytując Wikipedię: „W telefonii użytkowy zakres częstotliwości leży między 300 Hz a 3400 Hz". Innymi słowy, modemy musiały pracować w bardzo ograniczonym paśmie (około 3 kHz) na analogowym kanale z dość dużymi zakłóceniami. Pierwszy komercyjny modem wypuczono do sprzedaży pod koniec lat 50. ubiegłego wieku. Pozwalał on na transmisję danych na poziomie 100 bitów na sekundę. Przez lata prędkość ta stopniowo zwiększała się. Na końcu ery modemów (w późnych latach 90.) urządzenia oferowały już prędkości rzędu 56 kbit/s.

Web Audio API pozwala na generowanie i przetwarzanie dźwięku w JavaScripcie. Możemy zatem użyć go do stworzenia aplikacji działającej jak modem. Powietrze w naszym pokoju spełniałoby rolę analogowej linii telefonicznej. Ważną różnicą jest fakt, że obecny sprzęt ma dużo lepszą jakość. Pokrywa on w całości zakres dźwięków słyszalnych przez człowieka (w teorii częstotliwości pomiędzy 20 Hz a 20 kHz). Oznacza to, że mamy dużo większe pasmo do dyspozycji. Naszym celem jest przesłanie między dwoma komputerami danych binarnych nawet przy udziale niepożądanych zakłóceń, takich jak biały szum, muzyka czy rozmowa. Dodatkowo wszystkie operacje na próbkach audio zaimplementujemy samodzielnie. Pozwoli nam to zrozumieć od podstaw cały proces filtrowania sygnału. Jakich prędkości oczekujemy? Tak naprawdę nie chodzi tu o prędkość. Mogłoby być to nawet 8 bitów na sekundę. Prędkości na poziomie 64 bit/s pozwoliłyby przesyłać w kilka sekund np. numery telefonów czy adresy stron bez użycia maila czy komunikatora. Możliwe byłoby stworzenie prostego czata, który nie wymaga żadnego połączenia Internetowego.

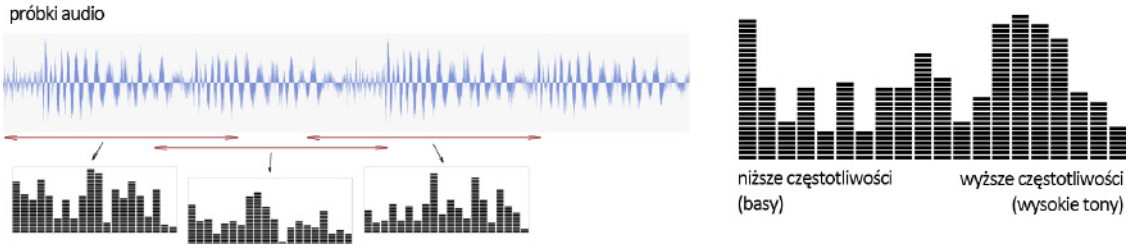
By stworzyć aplikację tego typu od podstaw, musimy zaprzęgnąć techniki cyfrowego przetwarzania sygnałów (ang. *Digital Signal Processing*, DSP). Pierwszą rzeczą, od jakiej warto zacząć, jest Dyskretna Transformata Fouriera (ang. *Discrete Fourier Transform*, DFT).

WPROWADZENIE DO DTF

Czy kiedykolwiek zastanawialiśmy się, jak to możliwe, że tak wiele bezprzewodowych urządzeń jest w stanie komunikować się jednocześnie bez żadnych zakłóceń? Na przykład możemy używać sieci Wi-Fi (2.4 GHz), telefonu komórkowego LTE (2600 MHz) oraz oglądać programy telewizyjne DVB-T (około 600 MHz) w tym samym czasie. Jedną z odpowiedzi jest fakt, że wszystkie te urządzenia pracują na różnych częstotliwościach fal elektromagnetycznych. Kiedy musimy radzić sobie z częstotliwościami, z pomocą przychodzi Transformata Fouriera. Dla sygnałów cyfrowych będzie to Dyskretna Transformata Fouriera. Co nam ona daje? Otóż transformata ta potrafi zamienić sygnał reprezentowany w dziedzinie czasu na dziedzinę częstotliwości. Innymi słowy, możemy wyciągnąć częstotliwości, które tworzą sygnał zmieniający się w czasie. Pozwala to np. skupić się tylko na wąskim zakresie, pomijając całą resztę spektrum. Transformata daje nam zatem możliwość filtrowania tylko tego, co nas interesuje.

Chwileczkę… czy nie mieliśmy czasem skupić się na dźwięku? Oczywiście tak, jednak generalnie w obydwu przypadkach chodzi o częstotliwość. Nie ma znaczenia, czy są to fale radiowe (3 kHz – 300 GHz) czy fale akustyczne słyszalne przez człowieka (20 Hz – 20 kHz). W obydwu przypadkach sygnał może być zdigitalizowany przez próbkowanie go w miarę upływu czasu, zatem algorytm jest ten sam.

W dzisiejszych czasach dźwięk nie jest już powszechnie używany do przesyłania informacji, jednak ciągle możemy znaleźć inne zastosowania DFT. Na przykład odtwarzacze muzyczne bardzo często pokazują na swoich wyświetlaczach wynik działania transformaty w postaci pasków equalizera. Z tych ładnie wyglądających „skaczących paseczków" możemy odczytać, jak głośny jest każdy z zakresów częstotliwości słuchanej piosenki. Na przykład możemy odczytać, jak szybki jest rytm basu, nawet bez zakładania słuchawek (Rysunek 1).



Rysunek 1. Piosenka oraz paski equalizera

SZYBKIE ORAZ WOLNE METODY OBLICZANIA DTF

Aktualnie najszybszym algorytmem obliczania Dyskretnej Transformaty Fouriera jest FFT (ang. *Fast Fourier Transform*). Jest on obecny niemal w każdej nowoczesnej technice bezprzewodowej komunikacji. Dla przykładu – pracuje on, gdy korzystamy z Internetu na telefonie obsługującym LTE, przesyłamy pliki poprzez sieć Wi-Fi czy też oglądamy program z naziemnej telewizji cyfrowej. Algorytm ten jest bardzo szybki, jednak dość trudny do zrozumienia. Ideą tego artykułu jest stworzenie prostego systemu przetwarzania sygnałów od podstaw, dlatego skupimy się na dużo prostszej i bardziej intuicyjnej metodzie. Ostatecznie wyniki obydwu rozwiązań będą podobne. Jest jednak jedna znacząca wada – ceną prostoty jest bardzo niska wydajność. W czasie rzeczywistym, bez zbytniego obciążania procesora, silnik JS przeglądarki jest w stanie przeanalizować tylko kilka częstotliwości. Dla porównania w metodzie FFT tych częstotliwości będą tysiące.

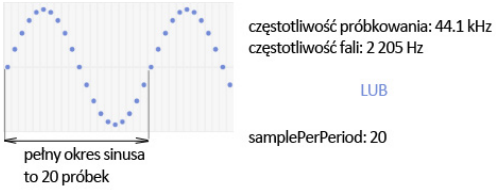
INTUICYJNA I PROSTA METODA OBLICZANIA DTF

Zanim zaczniemy, warto zmienić nieco sposób, w jakim wyrażamy częstotliwość fali. Zamiast używać wartości częstotliwości wyrażonej w Hertzach, możemy posłużyć się liczbą próbek mieszczących się w jednym pełnym okresie (`samplePerPeriod`). Pozwoli nam to w dalszych przykładach całkowicie zrezygnować z używania wartości częstotliwości podawanej w Hertzach. Jest ona zawsze powiązana z dodatkowym parametrem – częstotliwością próbkowania. Jako że pracować będziemy na tablicach próbek audio, ta odmienna reprezentacja ułatwi nam życie (Rysunek 2).

Do dalszych eksperymentów posłużymy się sygnałem zbudowanym z trzech przebiegów sinusoidalnych. Każdy z nich (Sinus A, Sinus B oraz Sinus C) ma inną częstotliwość. Gdy wyrazimy ją w próbkach na okres (`samplePerPeriod`), będzie to kolejno 28, 20 oraz 16. Dla dociekliwych przebiegi te można także wyrazić w Hertzach, posługując się wzorem `frequencyInHertz = sampleRate / samplePerPeriod`. Zakładając standardowe próbkowanie występujące np. w większości plików mp3 (44.1 kHz), otrzymamy kolejno 1575 Hz, 2205 Hz oraz 2756.25 Hz (Rysunek 3).

Gdy spojrzymy na wynikowy sygnał, jest bardzo ciężko stwierdzić, jakie częstotliwości są w nim zawarte. Trudno nawet powiedzieć, ile przebiegów sinusoidalnych zostało zsumowanych, by otrzymać sygnał o takim wyglądzie. Jak więc możemy wydobyć częstotliwości składowe, znając tylko sygnał wynikowy? Pierwszym krokiem jest zebranie odpowiedniej liczby próbek, których użyjemy do dalszych działań. Ten krok nazywamy okienkowaniem. Można go traktować także jako klatkę animacji, ponieważ na końcu pokazane zostaną częstotliwości sygnału w tym konkretnym momencie. W praktycznych zastosowaniach chcemy często wychwycić konkretną częstotliwość występującą przez krótki okres czasu. Nie miałoby więc większego sensu wykonanie DTF na wszystkich próbkach bufora naraz. Po pierwsze, próbek byłoby za dużo, po drugie – zmiany częstotliwości zostałyby zlane. Musimy zatem podzielić bufor na części i transformować każdą z nich z osobna. Często sąsiadujące okienka zachodzą na siebie tak, by nie tracić żadnej partii całego sygnału. W naszym przypadku okno o rozmiarze 1024 próbek będzie wystarczające (Rysunek 4).

W następnym kroku musimy na naszych próbkach zastosować funkcję okna. Jej celem jest łagodne „wygładzenie" prawej i lewej strony okna przy zachowaniu jego środka. Jest to bardzo ważne, ponieważ naszym zadaniem jest dekompozycja sygnału w przebiegi sinusoidalne, które go tworzą. Problem w tym, że nie wszystkie przebiegi mieszczą się w zadanym oknie w taki sposób, że wielokrotność okresów jest liczbą całkowitą. Na Rysunku 5 tylko ostatni przebieg spełnia ten warunek.

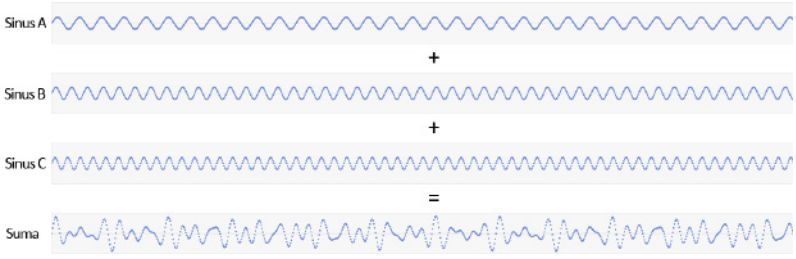


częstotliwość próbkowania: 44.1 kHz
częstotliwość fali: 2 205 Hz

LUB

samplePerPeriod: 20

Rysunek 2. Dwa sposoby reprezentacji częstotliwości fali



Rysunek 3. Składowe sygnału testowego

Rysunek 4. Próbki z bufora, które trafiły do okna

Bez zastosowania funkcji okna wszystkie nie w pełni mieszczące się przebiegi zaowocowałyby powstaniem efektu nazywanego wyciekiem widma. W rezultacie główne częstotliwości tworzące sygnał nie byłyby wyraźnie widoczne. Rysunek 6 pokazuje, jak wygląda przykładowa funkcja okna (w środku) oraz jak zmieniają się nasze próbki po jej zastosowaniu. W efekcie oba krańce okna zostały łagodnie wyzerowane.

Wyniki działania DTF pokazujemy najczęściej na wykresie widma amplitudowego (Rysunek 7). Każdy pionowy pasek nazywamy prążkiem widmowym (ang. *frequency bin*), który mówi, „jak wiele" powiązanej z nim częstotliwości znajduje się w sygnale z naszego okna. Załóżmy, że chcemy zobaczyć widmo sygnału dla zakresu `samplePerPeriod` od 10 do 50. By rozdzielczość była zadowalająca, ustalmy liczbę prążków na 160. Oznacza to, że będą one oddalone od siebie o wartość `samplePerPeriod` równą 0.25 (50-10/160=0.25). Długości prążków to wartości zmieniające się w bardzo szerokim zakresie. Niektóre z nich mogą być równe 0.1, podczas gdy inne mogą być rzędu 0.000001. Dla ukazania pełnej zmienności wartości oś pionowa wyrażona jest w decybelach. Wartość 0.1 będzie równa -10 decybelom, natomiast wartość 0.000001 będzie równa -60 decybelom.

Im niżej na osi pionowej, tym wartości maleją. -60 dB będą miały fale o znacznie mniejszej amplitudzie niż np. -5 dB.

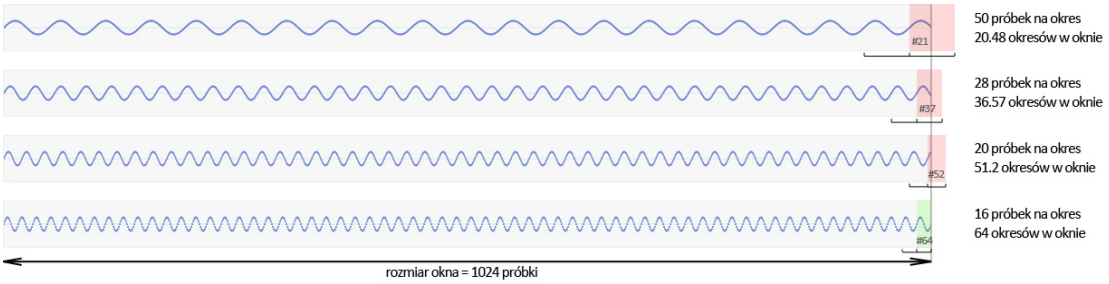
Na wykresach widma najczęściej wyższe częstotliwości znajdują się po prawej, a niższe po lewej stronie. W naszym przypadku niższa wartość `samplePerPeriod` oznacza wyższą częstotliwość, ponieważ w okresie sinusa jest mniej próbek – innymi słowy, jest on bardziej upakowany, więc częstotliwość jest większa. To jest powód, dla którego musimy umieścić wyższe wartości `samplePerPeriod` po lewej stronie, a zakończyć niższymi wartościami po prawej stronie.

Dla dociekliwych – wykres widma pokaże częstotliwości z zakresu od 882 Hz do około 4302 Hz (zakładając częstotliwość próbkowania 44.1 kHz).

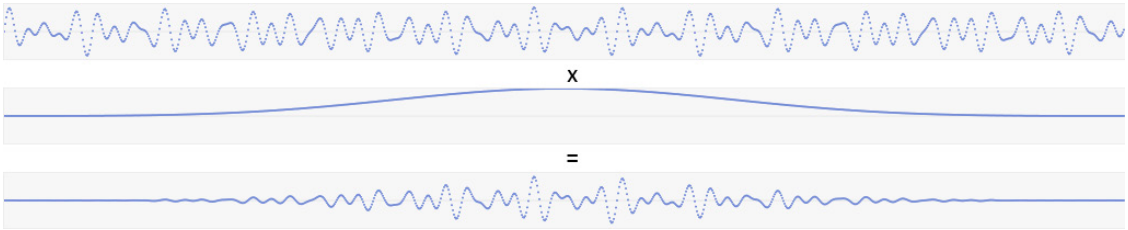
Użycie `samplePerPeriod` zamiast wartości częstotliwości w Hertzach wpłynie na oś poziomą wykresu widma. Konwersja ta nie jest liniowa, dlatego nasze prążki nie będą oddalone od siebie o taką samą wartość Hertzów. Zamiast tego będą one oddalone o taką samą wartość `samplePerPeriod`. Otrzymany wykres będzie zatem wyglądać nieco inaczej, niż tradycyjne widmo wygenerowane dla tego samego zakresu. Lewa strona będzie nieco bardziej rozciągnięta w poziomie niż prawa. Dla potrzeb tego artykułu nie stanowi to problemu.

Ok, zacznijmy najciekawszą część. Jak obliczyć Dyskretną Transformatę Fouriera na próbkach naszego okna? Musimy wykonać ten sam algorytm dla każdej częstotliwości (mamy ich 160), który brzmi:

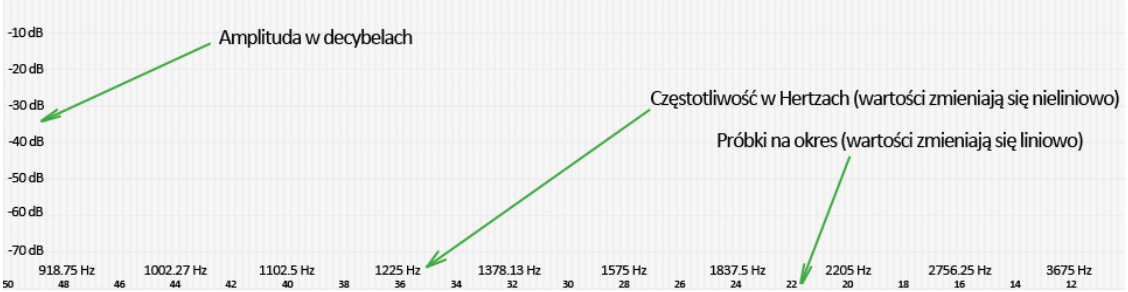
By otrzymać szczegóły jednej z częstotliwości zawartej w sygnale, musimy przeiterować przez wszystkie próbki naszego okna. W każdej iteracji musimy stworzyć dwuwymiarowy wektor jednostkowy. Jego punktem zaczepienia będzie zawsze początek układu współrzędnych, a długością, jak na wektor jednostkowy przystało, wartość jeden. Tylko kierunek będzie zależny od aktualnej iteracji. Gdybyśmy pokazali te wektory na animacji, zobaczylibyśmy, że zataczają one okręgi (jak wskazówki zegara) o okresie



Rysunek 5. Jak różne częstotliwości mieszczą się w oknie

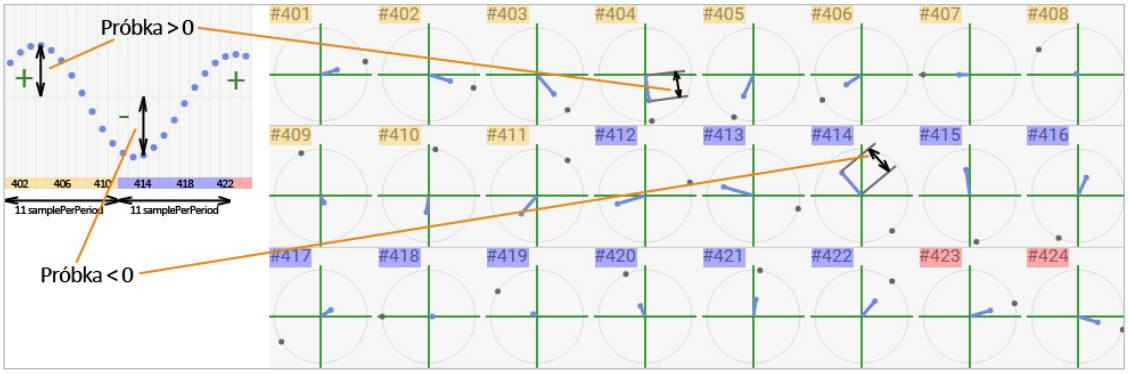


Rysunek 6. Zastosowanie funkcji okna

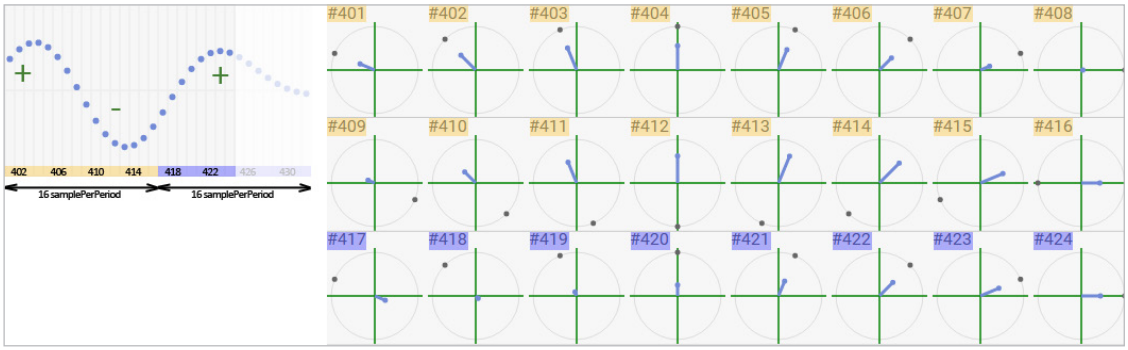


Rysunek 7. Pusty wykres widma amplitudowego

Rysunek 8. Część okna, która zostanie pokazana szczegółowo



Rysunek 9. Szczegóły 24 z 1024 przebiegów pętli (samplePerPeriod równe 11)



Rysunek 10. Szczegóły 24 z 1024 przebiegów pętli (samplePerPeriod równe 16)

związanym z aktualnie badaną częstotliwością. W każdej iteracji musimy pomnożyć nasz wektor jednostkowy przez wartość odpowiadającej mu próbki. Jako że wartości próbek zawierają się w przedziale od -1 do +1, ta operacja może skrócić długość naszego wektora. W przypadku ujemnych wartości wektor może zostać obrócony o 180 stopni. Z wszystkich iteracji otrzymamy tyle wektorów pomnożonych przez wartość próbki, ile wynosi rozmiar okna (w naszym przypadku 1024). Gdy zsumujemy je wszystkie ze sobą, otrzymamy również wektor dwuwymiarowy. Ostatnim krokiem jest normalizacja długości naszego finalnego wektora. Wystarczy podzielić go przez liczbę próbek znajdujących się w oknie.

Jakie informacje niesie nasz finalny wektor? Otóż mówi on nam wszystko o obecności danej częstotliwości w sygnale z naszego okna.

Długość finalnego wektora to amplituda fali o częstotliwości, którą analizujemy. Właśnie ta wartość cechuje się dużą zmiennością, dlatego należy ją zamienić na decybele.

Jeżeli wyznaczymy kąt (zgodnie z ruchem wskazówek zegara) między dodatnią częścią osi Y a finalnym wektorem, będzie on równy przesunięciu fazowemu naszej fali względem początku okna. W dalszych przykładach dowiemy się, czym dokładnie jest to przesunięcie.

Jako że nasz wykres nazywamy widmem amplitudowym, do rysowania prążków posłużymy się tylko długością wektorów finalnych wyrażoną w decybelach.

W analizie fourierowskiej nasze dwuwymiarowe wektory to tak naprawdę liczby zespolone. Oś X to część rzeczywista, a oś Y to część urojona. Dla prostoty możemy je jednak traktować jak zwykłe wektory 2D.

Na naszym wykresie widma zdecydowaliśmy się mieć 160 prążków. Rozmiar okna został wcześniej ustalony na 1024 próbki. Oznacza to, że do stworzenia kompletnego wykresu widma musimy wykonać 163 840 iteracji (1024*160). Przy zwiększaniu rozdzielczości lub/i rozmiaru okna ta wartość rośnie bardzo szybko. To jest powód, dla którego ten algorytm jest ultra wolny.

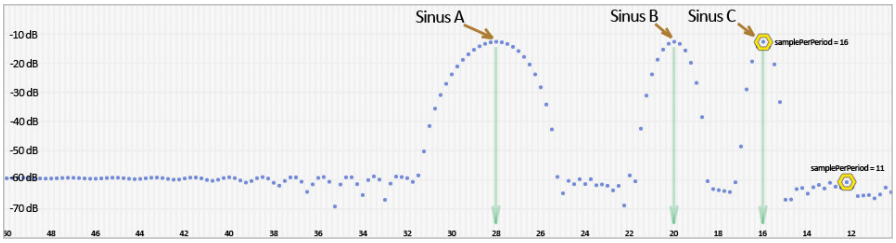
PRZYKŁADY

Przykłady są zawsze lepsze niż tysiąc słów. Nasze przebiegi sinusoidalne mają wartość `samplePerPeriod` równą kolejno 28, 20 oraz 16. Weźmy dla przykładu jeden „losowy” `samplePerPeriod` równy 11. Ta częstotliwość nie jest obecna w naszym sygnale, więc możemy oczekiwać, że wartość amplitudy tej częstotliwości będzie niska.

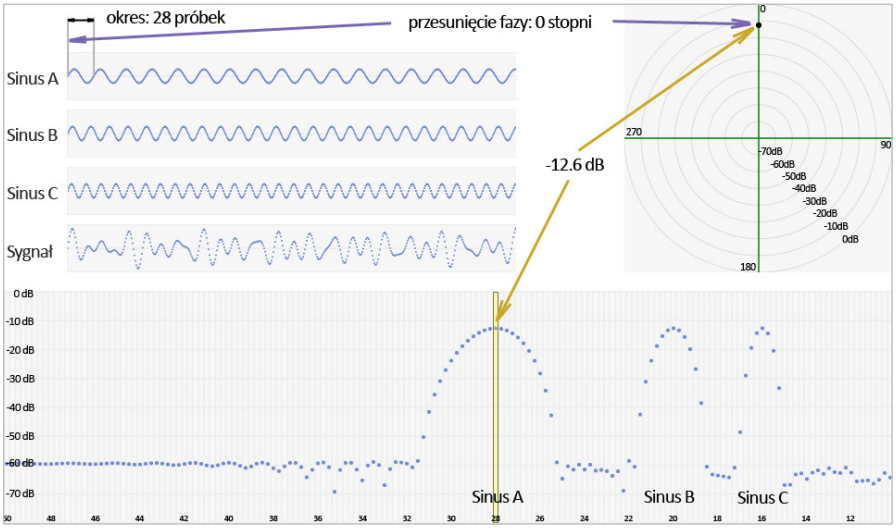
By to sprawdzić, musimy przeiterować przez wszystkie 1024 próbki naszego okna. Niestety ta liczba jest zbyt duża, by pokazać wszystko szczegółowo. Pokażmy zatem tylko 24 iteracje ze środka okna, ponieważ tam zmiany wartości sygnału są największe. Oczywiście „pod maską” obliczymy wszystkie 1024 próbki. Kolorem żółtym zaznaczono iteracje od 401 do 424, które zostaną pokazane szczegółowo (Rysunek 8).

Po zbliżeniu widać wyraźnie, jak wartość każdej próbki wpływa na powiązany z nią wektor jednostkowy (Rysunek 9).

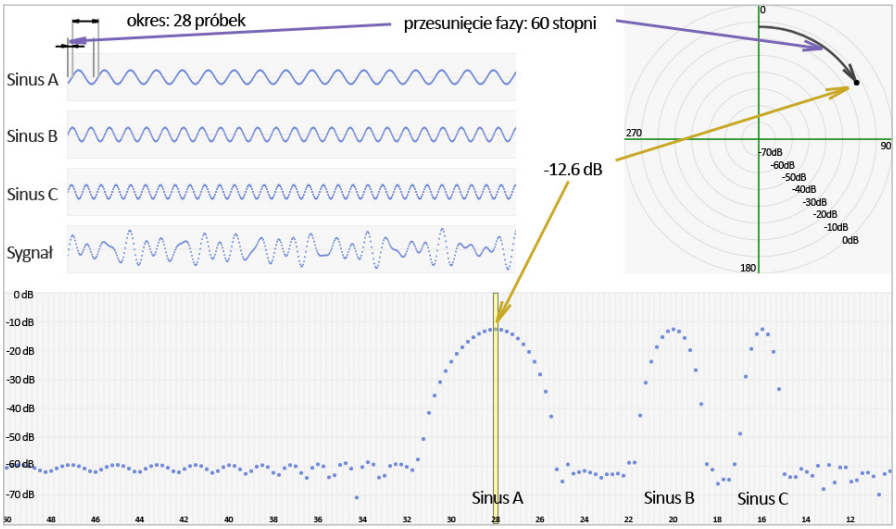
Ciemny punkt to koniec wektora jednostkowego mającego swój punkt zaczepienia w środku układu współrzędnych. Linia wektora została pominięta, by nie zaciemniać niebieskiego wek-



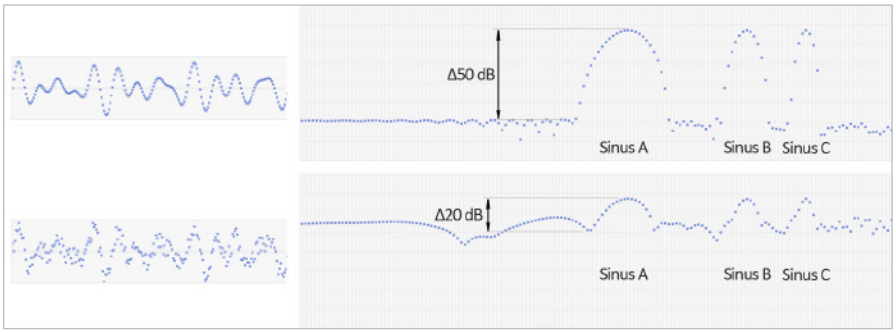
Rysunek 11. Wykres widma amplitudowego



Rysunek 12. Diagram konstelacji – Sinus A bez przesunięcia fazowego względem początku okna



Rysunek 13. Diagram konstelacji – Sinus A przesunięty względem początku okna



Rysunek 14. Porównanie sygnału czystego oraz zakłóconego

tora, który jest bardziej ważny. Niebieski wektor natomiast jest wynikiem pomnożenia wektora jednostkowego przez wartość próbki. W przykładowym prążku badamy falę o wartości `samplePerPeriod` równej 11. Jak widać, ciemna kropka zatacza pełny krąg co każde 11 iteracji. Nasze okno składa się z 1024 próbek. Oznacza to, że ciemna kropka (koniec wektora jednostkowego lub, jak kto woli, liczba zespolona) wykonuje około 93 pełnych obiegów wokół środka układu współrzędnych (1024/11). Generalnie im więcej obiegów, tym lepsze rezultaty otrzymamy. Patrząc tylko na dwa pełne obiegi, widać, że długości i kierunki niebieskich wektorów są w gruncie rzeczy przypadkowe.

Jeżeli częstotliwość, którą sprawdzamy, nie jest obecna w sygnale, spowoduje to powstanie wielu wektorów zwróconych w różnych kierunkach. Po ich zsumowaniu zniosą się one wzajemnie, co sprawi, że długość finalnego wektora będzie mała.

Co jednak stanie się, gdy wybierzemy wartość `samplePerPeriod` równą częstotliwości jednego z naszych sinusów? Dla przykładu weźmy Sinus C, dla którego ta wartość jest równa 16 (Rysunek 10).

Teraz koniec naszego wektora jednostkowego (ciemna kropka) wykonuje pełny obieg w 16 iteracji. W naszym oknie będziemy mieć zatem 64 pełnych rotacji (1024/16). Teraz większość długich niebieskich wektorów wydaje się być zwrócona w jednym kierunku (w górę lub, jak kto woli, na godzinę dwunastą). Możemy powiedzieć, że „wyłapały" one coś z naszego sygnału.

Jeżeli częstotliwość, którą sprawdzamy, jest obecna w sygnale, spowoduje to powstanie wielu wektorów skierowanych w tym samym kierunku. Porównać to możemy z huśtawką. Mała siła o odpowiedniej częstotliwości zwiększy amplitudę wychyleń huśtawki. Długość finalnego wektora będzie zatem dużo większa niż w poprzednim przykładzie.

WYKRES WIDMA AMPLITUDOWEGO ORAZ DIAGRAM KONSTELACJI

Gdy przeiterujemy przez wszystkie 160 prążków, otrzymane wartości pozwolą nam narysować wykres widma (Rysunek 11). Jak widzimy, prążki w okolicach częstotliwości zbliżonych do naszych trzech sinusów stworzyły „górki".

Na wykresie z Rysunku 11 zaznaczono także dwie częstotliwości z poprzednich

przykładów. Jak widzimy, amplituda częstotliwości o `samplePerPeriod` 11 znajduje się na poziomie szumu (około -60 dB). Odmienna sytuacja jest w przypadku częstotliwości o `samplePerPeriod` 16. Tam wartość amplitudy wynosi około -13 dB. Z układu prążków widzimy, że jest to lokalny szczyt. Oznacza to, że w sygnale wejściowym na tej częstotliwości obecna jest sinusoida o znacznej amplitudzie. Jest to prawda, ponieważ nasz Sinus C ma dokładnie tą częstotliwość.

Wykres widma jest w większości przypadków wystarczający. Należy jednak pamiętać, że jest to graficzna reprezentacja jedynie długości dwuwymiarowych wektorów wyznaczonych dla każdego prążka (lub, jak kto woli, wartości bezwzględnej liczby zespolonej). Jest to zatem „spłaszczona" wersja tego, co daje nam Dyskretna Transformata Fouriera. Oprócz wartości amplitudy posiadamy także informację o fazie fali. Jeżeli interesują nas obie właściwości, wtedy z pomocą przychodzi diagram konstelacji. Pokazuje on jednak informację tylko o jednej wybranej przez nas częstotliwości, a nie, jak w przypadku wykresu widma, wszystkich naraz.

Z diagramu konstelacji możemy odczytać wartość amplitudy w decybelach oraz przesunięcie fazowe fali, którą badamy. Im dalej położony będzie punkt od środka układu współrzędnych, tym sygnał nadawany na tej częstotliwości (np. nasze dane binarne) będzie silniejszy. Pozycja puntu na okręgu mówi natomiast o przesunięciu fazowym. Punkty na górze (godzina dwunasta) mają przesunięcie fazowe równe 0 stopni (lub 360 stopni, jako że sinus jest okresowy). Wartości rosną zgodnie z ruchem wskazówek zegara, zatem po prawej stronie będziemy mieć przesunięcie równe 90 stopni (godzina trzecia), natomiast na dole 180 stopni (godzina szósta) itd.

Jeżeli sinus zawarty w sygnale z okna nie będzie miał żadnego przesunięcia fazowego względem jego początku, to punkt na diagramie konstelacji będzie położony na godzinie dwunastej. Żółty marker pokazuje, który prążek powiązany jest z punktem na diagramie (Rysunek 12).

Jeżeli jednak nieco przesuniemy w czasie nasz przebieg „Sinus A", spowoduje to obrócenie punktu na diagramie konstelacji (Rysunek 13).

Jak widać, wartość amplitudy (odległość punktu od początku układu współrzędnych na diagramie konstelacji) nie zmieniła się zbytnio. Jedynie punkt związany z sinusem A został obrócony, ponieważ 1024 wektory zsumowane w pętli dały finalny wektor zwrócony w nieco inną stronę. Gdy przyjrzymy się prążkom na wykresie widma amplitudowego, także niewiele się zmieniło. Jest to zgodnie z intuicją, ponieważ zmieniliśmy tylko fazę częstotliwości składowej sygnału.

ZRÓBMY HAŁAS!

Prawdziwe sygnały nigdy nie są idealne. Będą one zawierać zakłócenia związane z odbiciami, rozmową ludzi itp. Sprawdźmy zatem najprostszy do zasymulowania typ zakłócenia nazywany „białym szumem". By tego dokonać, jedyne, co musimy zrobić, to tuż przed zastosowaniem DTF do każdej próbki dodać losową wartość (Listing 1).

Listing 1. Symulacja „białego szumu"

```
// dodaje losową wartość z przedzialu od -0.3 do 0.3
whiteNoiseAmplitude = 0.3;
sample += (-1 + 2 * Math.random()) * whiteNoiseAmplitude;
```

Jak widzimy, nasz sygnał w dziedzinie czasu jest teraz bardzo zniekształcony (Rysunek 14). Gdy spojrzymy na wykres widma, ciągle jesteśmy w stanie zobaczyć nasze trzy przebiegi, jednak różnica między najwyższymi prążkami a szumem znacząco zmalała.

IMPLEMENTACJA W JAVASCRIPTCIE

W tej sekcji artykułu możemy znaleźć implementację tego, o czym była mowa do tej pory.

Listing 2. Implementacja algorytmu obliczającego Dyskretną Transformatę Fouriera

```
function getFrequencyDetails(timeDomain, samplePerPeriod) {
  var
    amplitude, amplitudeDecibel, phase,
    unitReal, unitImm, i, sample, radian,
    windowSize = timeDomain.length,
    decibelLimit = -80,
    real = 0,
    imm = 0;

  for (i = 0; i < windowSize; i++) {
    sample = timeDomain[i];

    // tworzenie wektora jednostkowego
    radian = 2 * Math.PI * i / samplePerPeriod;
    unitReal = -Math.cos(radian);
    unitImm = Math.sin(radian);

    // wektor jednostkowy pomnożony przez wartość próbki
    real += unitReal * sample;
    imm += unitImm * sample;
  }

  // normalizacja finalnego wektora
  real /= windowSize;
  imm /= windowSize;

  // długość finalnego wektora to amplituda sinusa
  amplitude = Math.sqrt(real * real + imm * imm);
  amplitudeDecibel = 10 * Math.log(amplitude) / Math.LN10;
  amplitudeDecibel = amplitudeDecibel < decibelLimit
    ? decibelLimit : amplitudeDecibel;

  // kąt między dodatnią częścią osi Y a finalnym wektorem
  // to przesunięcie fazowe sinusa względem początku okna
  phase = findUnitAngle(real, imm);

return {
    amplitudeDecibel: amplitudeDecibel,
    phase: phase
  };
}

function computeDFT(timeDomain, freqBinSPPMax,
  freqBinSPPMin, freqBinSize) {
  var
    frequencyDomain, frequencyDetails,
    i, step, samplePerPeriod;

  frequencyDomain = [];
  step = (freqBinSPPMax - freqBinSPPMin) / freqBinSize;
  for (i = 0; i < freqBinSize; i++) {
    samplePerPeriod = freqBinSPPMax - i * step;
    frequencyDetails = getFrequencyDetails(
      timeDomain, samplePerPeriod
    );
    frequencyDomain.push(frequencyDetails);
  }

  return frequencyDomain;
}

function findUnitAngle(x, y) {
  var
    length = Math.sqrt(x * x + y * y),
    lengthLimit = 0.000001,
    quarter,
    angle;
```

```
  length = (length < lengthLimit) ? lengthLimit : length;
  quarter = (x >= 0) ? (y >= 0 ? 0 : 1) : (y < 0 ? 2 : 3);
  switch (quarter) {
    case 0:
      angle = Math.asin(x / length);
      break;
    case 1:
      angle = Math.asin(-y / length) + 0.5 * Math.PI;
      break;
    case 2:
      angle = Math.asin(-x / length) + Math.PI;
      break;
    case 3:
      angle = Math.asin(y / length) + 1.5 * Math.PI;
      break;
  }

  return angle / (2 * Math.PI); // kąt w przedziale <0, 1)
}

function blackmanNuttall(n, N) {
  var x = Math.PI * n / (N - 1);

  return 0.3635819
    - 0.4891775 * Math.cos(2 * x)
    + 0.1365995 * Math.cos(4 * x)
    - 0.0106411 * Math.cos(6 * x);
}

function generateSine(samplePerPeriod, amplitude,
  degreesPhaseOffset, sample) {
  var
    unitPhaseOffset = degreesPhaseOffset / 360,
    x = (sample / samplePerPeriod - unitPhaseOffset);

  return amplitude * Math.sin(2 * Math.PI * x);
}
```

Po lekturze wstępu teoretycznego kod z Listingu 2 powinien być w miarę prosty do zrozumienia. Cztery pierwsze funkcje stanowią bazę do obliczania Dyskretnej Transformaty Fouriera, natomiast ostatnia służy do generowania przebiegów sinusoidalnych. Magiczny wzór z funkcji `blackmanNuttall` został zaczerpnięty z Wikipedii z artykułu o funkcjach okna (https://en.wikipedia.org/wiki/Window_function).

Wygenerujmy zatem próbki sygnału zawierającego nasze trzy sinusy z poprzednich przykładów i wykonajmy na nich DTF (Listing 3).

Listing 3. Generowanie próbek sygnału w dziedzinie czasu i obliczenie DTF

```
var
  i, sample, sampleProcessed, frequencyDomain,
  freqBinSPPMax, freqBinSPPMin, freqBinSize,
  whiteNoiseAmplitude = 0,
  windowSize = 1024,
  timeDomain = [];

for (i = 0; i < windowSize; i++) {
  sample = 0;

  // Sinus A: samplePerPeriod 28, amplituda 0.3, faza 0
  sample += generateSine(28, 0.3, 0, i);
  // Sinus B: samplePerPeriod 20, amplituda 0.3, faza 0
  sample += generateSine(20, 0.3, 0, i);
  // Sinus C: samplePerPeriod 16, amplituda 0.3, faza 0
  sample += generateSine(16, 0.3, 0, i);

  // dodanie białego szumu
  sample +=
    (-1 + 2 * Math.random()) * whiteNoiseAmplitude;

  // zastosowanie funkcji okna
  sampleProcessed =
    sample * blackmanNuttall(i, windowSize);

  timeDomain.push(sampleProcessed);
}

// konfiguracja prążków
freqBinSPPMax = 50; // samplePerPeriod z lewej
freqBinSPPMin = 10; // samplePerPeriod z prawej
freqBinSize = 160; // liczba prążków

frequencyDomain = computeDFT(
  timeDomain, freqBinSPPMax, freqBinSPPMin, freqBinSize
);

// dziedzina czasu - tablica zawiera 1024 próbki
console.log(timeDomain.length); // --> 1024

// dziedzina częstotliwości - 160 obiektów o strukturze:
// { amplitudeDecibel: <float>, phase: <float> }
console.log(frequencyDomain.length); // --> 160
```

W przykładzie z Listingu 3 pominięto krok okienkowania – sygnał został bezpośrednio wpisany do tablicy okna. W prawdziwej aplikacji należałoby skopiować odpowiednią liczbę próbek z bufora wejściowego (np. mikrofonu). By otrzymywać zawsze takie same wyniki, wyłączony został „biały szum".

Przyjrzyjmy się, jakie wartości amplitudy otrzymaliśmy w okolicach naszych trzech sinusów. Spodziewamy się zobaczyć trzy skoki w okolicach `samplePerPeriod` o wartościach kolejno 28, 20 i 16. Do konwersji indeksu tablicy `frequencyDomain` na wartość `samplePerPeriod` i odwrotnie musimy posłużyć się wzorami przedstawionymi na Listingu 4.

Listing 4. Zamiana indeksu tablicy na samplePerPeriod i vice versa

```
/*
step = (freqBinSPPMax - freqBinSPPMin) / freqBinSize;
index = (freqBinSPPMax - samplePerPeriod) / step;
samplePerPeriod = freqBinSPPMax - step * index;
*/
```

Po obliczeniu odpowiednich indeksów tablicy możemy zweryfikować amplitudy w pobliżu naszych sinusów (Listing 5).

Listing 5. Wartości amplitudy w okolicach sinusów

```
function logAmplitude(index) {
  var
    freqDetails = frequencyDomain[index],
    amplitudeDecibel = freqDetails.amplitudeDecibel,
    result = Math.round(amplitudeDecibel * 100) / 100;

  console.log(result);
}

logAmplitude(87); // -12.81 | samplePerPeriod=28.25
logAmplitude(88); // -12.64 | samplePerPeriod=28.00 Sinus A
logAmplitude(89); // -12.82 | samplePerPeriod=27.75
// ...
logAmplitude(104); // -61.48 | samplePerPeriod=24.00
// ...
logAmplitude(119); // -13.32 | samplePerPeriod=20.25
logAmplitude(120); // -12.64 | samplePerPeriod=20.00 Sinus B
logAmplitude(121); // -13.35 | samplePerPeriod=19.75
// ...
logAmplitude(128); // -63.55 | samplePerPeriod=18.00
// ...
logAmplitude(135); // -14.30 | samplePerPeriod=16.25
logAmplitude(136); // -12.64 | samplePerPeriod=16.00 Sinus C
logAmplitude(137); // -14.41 | samplePerPeriod=15.75
```

Jak widać, największe amplitudy przypadają dokładnie tam, gdzie ich oczekiwaliśmy. Dla porównania wartości pomiędzy sinusami są bardzo niskie (około -60 decybeli). Na koniec sprawdźmy wartość przesunięcia fazowego (Listing 6).

Listing 6. Wartości fazy dla sinusów bez przesunięcia

```
function logPhase(index) {
  var
    freqDetails = frequencyDomain[index],
    phaseDegrees = Math.round(freqDetails.phase * 360),
    result = phaseDegrees % 360; // zakres <0, 360)

  console.log(result);
}

logPhase(88);  // 0 | Sinus A: generateSine(28, 0.3, 0, i);
logPhase(120); // 0 | Sinus B: generateSine(20, 0.3, 0, i);
logPhase(136); // 0 | Sinus C: generateSine(16, 0.3, 0, i);
```

Wszystkie fazy równe są zero, co pokrywa się z oczekiwaniami. Sprawdźmy zatem, co otrzymamy, gdy na etapie generowania dziedziny czasu nieco przesuniemy fazy sinusów względem początku okna (Listing 7).

Listing 7. Wartości fazy po przesunięciu sinusów względem początku okna

```
for (i = 0; i < windowSize; i++) {
  // ...
  // Sinus A: teraz przesunięcie fazowe wynosi 90
  sample += generateSine(28, 0.3, 90, i);
  // Sinus B: teraz przesuniecie fazowe wynosi 180
  sample += generateSine(20, 0.3, 180, i);
  // Sinus C: teraz przesuniecie fazowe wynosi 270
  sample += generateSine(16, 0.3, 270, i);
  // ...
}
// ...
logPhase(88);  // 90  | Sinus A
logPhase(120); // 180 | Sinus B
logPhase(136); // 270 | Sinus C
```

Jak widzimy, przesunięcie fazowe również zostało „odgadnięte" prawidłowo. Wartości te będą zbliżone nawet, gdy włączymy „biały szum", ustawiając `whiteNoiseAmplitude` na np. 0.3.

Powyższa implementacja dostępna jest na stronie projektu AudioNetwork (dft-simple). Dostępna jest także nieco bardziej rozbudowana wersja (dft-full) umożliwiająca eksperymentowanie z różnymi ustawieniami Dyskretnej Transformaty Fouriera:

- https://audio-network.rypula.pl/dft-simple
- https://audio-network.rypula.pl/dft-simple-src
- https://audio-network.rypula.pl/dft-full
- https://audio-network.rypula.pl/dft-full-src

PODSUMOWANIE

Opisany algorytm może nie jest optymalny, lecz relatywnie prosty w porównaniu z metodą nazywaną Szybką Transformatą Fouriera używającą trików matematycznych wykraczających poza zakres tego artykułu. Nasz cel został osiągnięty – jesteśmy w stanie wykryć główne częstotliwości, które tworzą sygnał, nawet gdy jest w nim wiele zakłóceń.

Nasuwa się pytanie – skoro opisany algorytm jest tak powolny w porównaniu z FFT, to czy jest jakieś praktyczne jego zastosowanie? Odpowiedź brzmi: i tak, i nie. Raczej nie nadaje się on do wyświetlania widma sygnału w czasie rzeczywistym. Na dzisiejszych sprzęcie w JavaScripcie możliwe jest przetworzenie maksymalnie kilkudziesięciu prążków. Takie osiągi przekładają się na niską rozdzielczość wykresu i w 100% obciążony procesor. Słaba wydajność nie stanowi jednak problemu przy prostej transmisji danych, gdzie możliwe jest użycie nawet tylko jednej częstotliwości. Cała reszta widma przestanie nam być potrzebna. W artykule omawialiśmy wykres składający się z 160 prążków. Na przykład przesyłając dane na 4 różnych częstotliwościach nośnych, zwiększylibyśmy wydajność 40 razy. Algorytm jest także łatwo skalowalny, więc obliczenia można dodatkowo przenieść do WebWorkerów, by wykorzystać potencjał procesorów wielordzeniowych.

Wracając do JavaScriptu – jak dostać się do próbek strumienia audio? Możemy tego dokonać poprzez Web Audio API. Pozwala ono w łatwy sposób odczytać dane z mikrofonu podłączonego do naszego komputera. Próbki przychodzą w paczkach w stałych odstępach czasu uzależnionych od rozmiaru paczki. Musimy jedynie napisać handler, który jest zwykłą funkcją. W parametrze mamy dostęp do tablicy o rozmiarach będących potęgą dwójki (256, 512, ..., 16384). Podobnie postępujemy, gdy chcemy wygenerować dźwięk i odtworzyć go na głośnikach. Różnica jest taka, że zamiast odczytywać – zapisujemy do tablicy ze zdarzenia.

Inne pytanie, które może nasunąć się po przeczytaniu tego artykułu – czy naprawdę musimy pisać wszystko od zera? Techniki DSP nie są nowością, więc powinno być wiele bibliotek gotowych do użycia. To prawda, jednak takie rozwiązanie zabrałoby całą zabawę. Można porównać to do samochodu. Możemy po prostu do niego wsiąść i odjechać. Jeśli jednak dodatkowo wiemy, jak on działa, to jest jeszcze lepiej. Jeżeli naprawdę interesują nas alternatywy, to jest nią AnalyzerNode z Web Audio API. Rozwiązanie to jest bardzo szybkie, ponieważ używa FFT. Niestety jego główną wadą jest fakt, że nie zwraca fazy analizowanych częstotliwości. Faza fali może zostać użyta w metodzie transmisji danych dużo bardziej odpornej na zakłócenia. Algorytm opisany w tym artykule daje nam więcej możliwości. Uniezależnia nas od zewnętrznych bibliotek oraz jest relatywnie prosty do zrozumienia. Unikamy dzięki temu „czarnego pudełka", które przetwarza próbki w nieznany nam magiczny sposób.

W następnej części artykułu znajdziemy opis zasady działania aplikacji umożliwiającej przesłanie danych binarnych między dwoma komputerami. Poznamy m.in. techniki modulacji cyfrowej oraz to, jak używać Web Audio API w praktyce. Algorytm Dyskretnej Transformaty Fouriera posłuży nam do przetwarzania w czasie rzeczywistym próbek pochodzących z naszego mikrofonu.

ROBERT RYPUŁA
robert.rypula@gmail.com

Pasjonat komputerów i programowania. Od wielu lat związany z aplikacjami webowymi. Obecnie zatrudniony w PGS Software na stanowisku Frontend Developer. Wcześniej doświadczenie zdobywał w firmie Okinet. Poza technologiami webowymi twórca m.in. aplikacji umożliwiającej rendering topografii całej Ziemi (OpenGL/Qt). Fan pisania własnych rozwiązań od zera wszędzie tam, gdzie jest to możliwe.