TECHNIKI PROGRAMOWANIA - projekt 3

Maciej Mazur 203723

Dawid Literski 203697

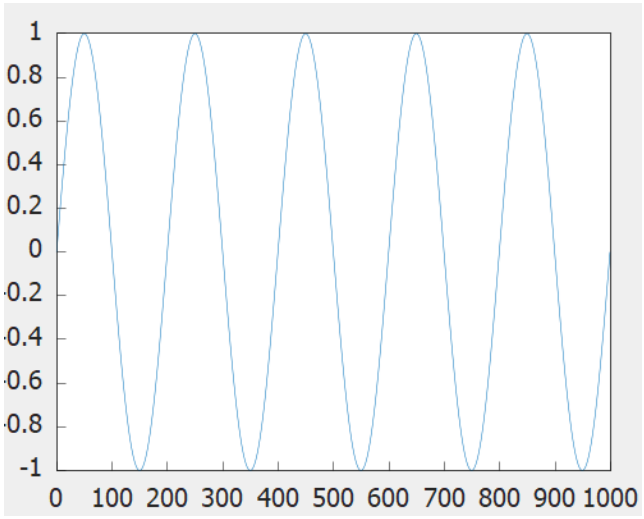
Biblioteki(cpp):pybind11, matplot , vector , cmath , complex , random;

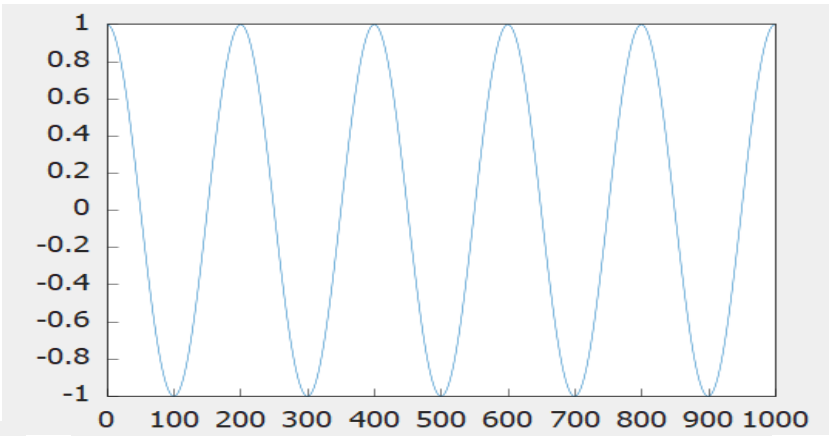
1. Wizualizacja sygnału z wykorzystaniem biblioteki matplotplusplus. + 4. Generowanie sygnałów o zadanej częstotliwości (sin, cos, prostokątny, piłokształtny).

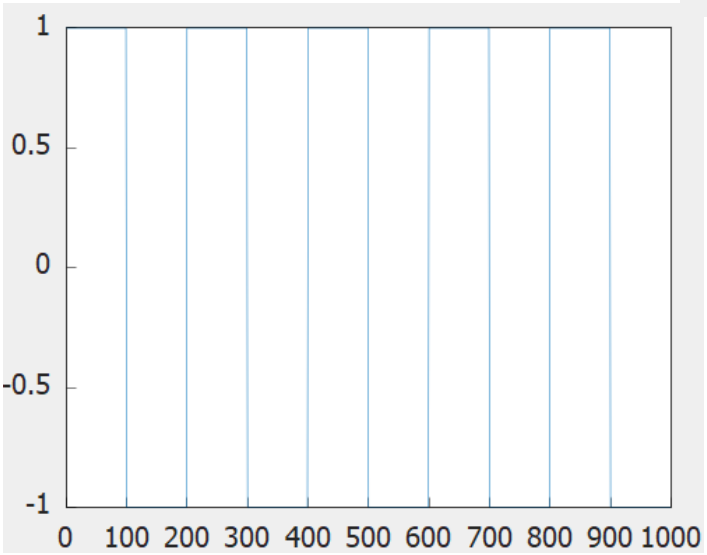
Wykorzystywanie mechanizmów z zadanie pierwszego

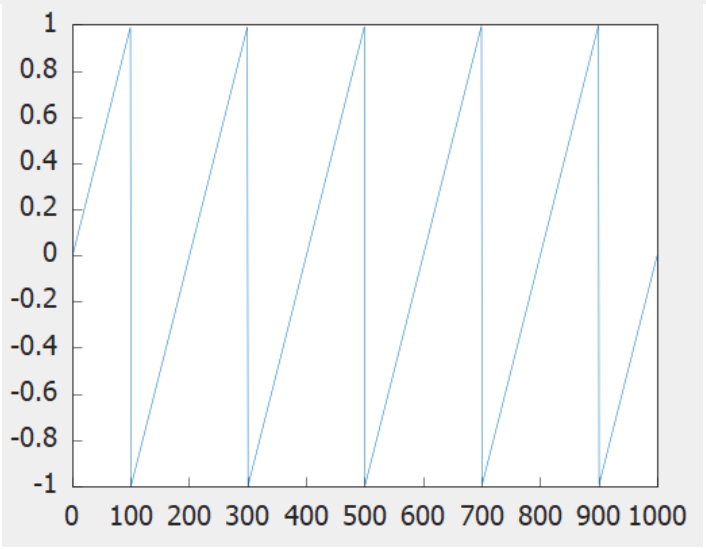
```python
# Parametry sygnału
fs = 1000       # Częstotliwość próbkowania (Hz)
f = 5           # Częstotliwość sygnału (Hz)
start = 0.0     # Początek sygnału (sekundy)
end = 1.0       # Koniec sygnału (sekundy)
samples = 1000 # Liczba próbek

print("Generowanie i rysowanie sygnałów:")

print("-> Sinusoida")
fala_sinus = example.generate_sine(f, start, end, samples)
example.plot_signal(fala_sinus)

print("-> Cosinusoida")
fala_cosinus = example.generate_cosine(f, start, end, samples)
example.plot_signal(fala_cosinus)

print("-> Prostokątny")
fala_kwadrat = example.generate_square(f, start, end, samples)
example.plot_signal(fala_kwadrat)

print("-> Piłokształtny")
fala_pila = example.generate_sawtooth(f, start, end, samples)
example.plot_signal(fala_pila)
```

```cpp
// Generowanie sygnałów (sin, cos, kwadrat, piłokształtny)
std::vector<double> generate_sine(double freq, double start, double end, int samples) {
    auto t = linspace(start, end, samples);
    std::vector<double> result(samples);
    for (int i = 0; i < samples; ++i)
        result[i] = sin(2 * M_PI * freq * t[i]);
    return result;
}

std::vector<double> generate_cosine(double freq, double start, double end, int samples) {
    auto t = linspace(start, end, samples);
    std::vector<double> result(samples);
    for (int i = 0; i < samples; ++i)
        result[i] = cos(2 * M_PI * freq * t[i]);
    return result;
}

std::vector<double> generate_square(double freq, double start, double end, int samples) {
    auto t = linspace(start, end, samples);
    std::vector<double> result(samples);
    for (int i = 0; i < samples; ++i)
        result[i] = sin(2 * M_PI * freq * t[i]) >= 0 ? 1.0 : -1.0;
    return result;
}

std::vector<double> generate_sawtooth(double freq, double start, double end, int samples) {
    auto t = linspace(start, end, samples);
    std::vector<double> result(samples);
    for (int i = 0; i < samples; ++i)
        result[i] = 2.0 * (t[i] * freq - floor(t[i] * freq + 0.5));
    return result;
}

// Rysowanie sygnału
void plot_signal(const std::vector<double>& y) {
    std::vector<double> x(y.size());
    for (size_t i = 0; i < y.size(); ++i)
        x[i] = i;
    plt::plot(x, y);
    plt::show();
}
```





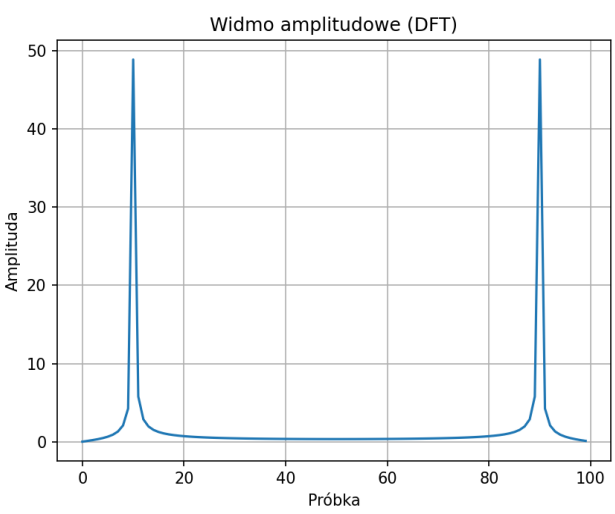


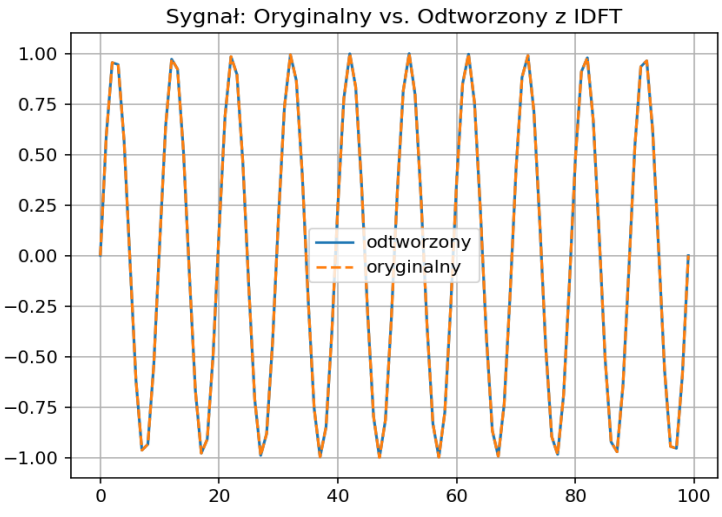


2. DFT i transformata odwrotna.

```python
# Przykładowy sygnał do DFT
signal = example.generate_sine(10, start, end, 100)

# DFT
print("-> DFT")
spectrum = example.dft(signal)

# Rysowanie widma
magnitudes = [abs(c) for c in spectrum]
plt.figure()
plt.plot(magnitudes)
plt.title("Widmo amplitudowe (DFT)")
plt.xlabel("Próbka")
plt.ylabel("Amplituda")
plt.grid()
plt.show()

# IDFT
print("-> IDFT")
restored = example.idft(spectrum)
plt.figure()
plt.plot(restored, label="odtworzony")
plt.plot(signal, label="oryginalny", linestyle='dashed')
plt.legend()
plt.title("Sygnał: Oryginalny vs. Odtworzony z IDFT")
plt.grid()
plt.show()
```

```cpp
// DFT i IDFT
std::vector<std::complex<double>> dft(const std::vector<double>& input) {
    size_t N = input.size();
    std::vector<std::complex<double>> output(N);
    for (size_t k = 0; k < N; ++k) {
        std::complex<double> sum = 0;
        for (size_t n = 0; n < N; ++n) {
            double angle = -2.0 * M_PI * k * n / N;
            sum += input[n] * std::complex<double>(cos(angle), sin(angle));
        }
        output[k] = sum;
    }
    return output;
}

std::vector<double> idft(const std::vector<std::complex<double>>& input) {
    size_t N = input.size();
    std::vector<double> output(N);
    for (size_t n = 0; n < N; ++n) {
        std::complex<double> sum = 0;
        for (size_t k = 0; k < N; ++k) {
            double angle = 2.0 * M_PI * k * n / N;
            sum += input[k] * std::complex<double>(cos(angle), sin(angle));
        }
        output[n] = sum.real() / N;
    }
    return output;
}
```



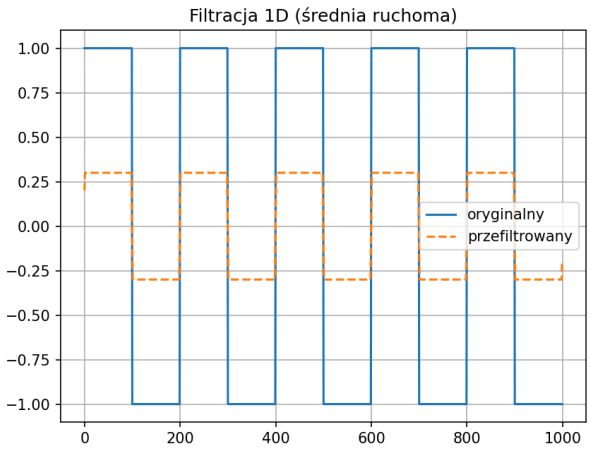


3. Filtracja 1D i 2D.

```python
# FILTRACJA 1D – np. wygładzanie prostokątnego
print("-> Filtracja 1D")
kernel = [1/10, 1/10, 1/10]  # prosty filtr uśredniający
filtered = example.convolve_1d(fala_kwadrat, kernel)
plt.figure()
plt.plot(fala_kwadrat, label="oryginalny")
plt.plot(filtered, label="przefiltrowany", linestyle='dashed')
plt.legend()
plt.title("Filtracja 1D (średnia ruchoma)")
plt.grid()
plt.show()
```

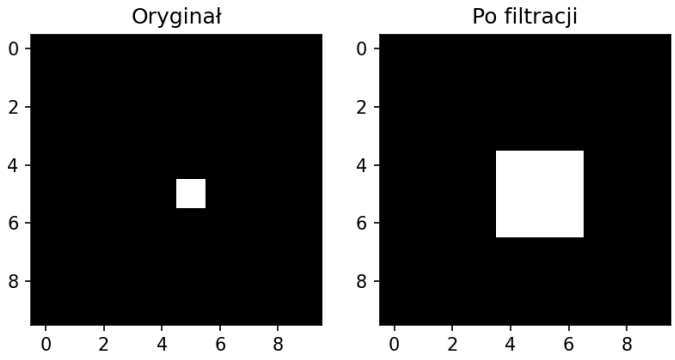
```cpp
// Filtracja 1D (konwolucja)
std::vector<double> filter_1d(const std::vector<double>& signal, const std::vector<double>& kernel) {
    int n = signal.size();
    int k = kernel.size();
    int half_k = k / 2;
    std::vector<double> result(n, 0.0);

    for (int i = 0; i < n; ++i) {
        double sum = 0.0;
        for (int j = 0; j < k; ++j) {
            int idx = i + j - half_k;
            if (idx >= 0 && idx < n)
                sum += signal[idx] * kernel[j];
        }
        result[i] = sum;
    }
    return result;
}
```

```cpp
std::vector<double> convolve_1d(const std::vector<double>& signal, const std::vector<double>& kernel) {
    int n = signal.size();
    int k = kernel.size();
    int pad = k / 2;
    std::vector<double> result(n, 0.0);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < k; ++j) {
            int idx = i + j - pad;
            if (idx >= 0 && idx < n) {
                result[i] += signal[idx] * kernel[j];
            }
        }
    }
    return result;
}
```




```python
# Przykładowa macierz obrazu 10x10 z "jasnym punktem" na środku
image = [[0]*10 for _ in range(10)]
image[5][5] = 100

# Jądro rozmywające (średnia z sąsiadów)
kernel = [[1/9]*3 for _ in range(3)]

# Filtracja
filtered = example.convolve_2d(image, kernel)

# Wizualizacja
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title("Oryginał")

plt.subplot(1, 2, 2)
plt.imshow(filtered, cmap='gray')
plt.title("Po filtracji")

plt.show()
```

```cpp
using Matrix = std::vector<std::vector<double>>;

std::vector<std::vector<double>> convolve_2d(const std::vector<std::vector<double>>& image,
                                             const std::vector<std::vector<double>>& kernel) {
    int rows = image.size();
    int cols = image[0].size();
    int krows = kernel.size();
    int kcols = kernel[0].size();
    int kcenterX = kcols / 2;
    int kcenterY = krows / 2;

    std::vector<std::vector<double>> result(rows, std::vector<double>(cols, 0.0));

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            double sum = 0.0;
            for (int m = 0; m < krows; ++m) {
                for (int n = 0; n < kcols; ++n) {
                    int x = j + n - kcenterX;
                    int y = i + m - kcenterY;
                    if (x >= 0 && x < cols && y >= 0 && y < rows) {
                        sum += image[y][x] * kernel[m][n];
                    }
                }
            }
            result[i][j] = sum;
        }
    }
    return result;
}
```



8. Zaszumianie sygnału (dodawanie sygnału losowego).

```cpp
// Dodawanie szumu Gaussowskiego do sygnału
std::vector<double> add_noise(const std::vector<double>& signal, double noise_stddev) {
    std::vector<double> noisy_signal(signal.size());
    std::default_random_engine generator(std::random_device{}());
    std::normal_distribution<double> distribution(0.0, noise_stddev);

    for (size_t i = 0; i < signal.size(); ++i) {
        noisy_signal[i] = signal[i] + distribution(generator);
    }
    return noisy_signal;
}
```

```
print("-> Zaszumianie sygnału (szum Gaussowski)")
noisy = example.add_noise(fala_sinus, 0.5)  # 0.5 to stddev szumu - może Pan zmieniać :)
example.plot_signal(noisy)
```