Student: Ghiorghe Sorana
Grupa: 30235

# Redimensionarea imaginilor (zoom and shrink)

Student: Ghiorghe Sorana
Grupa: 30235

# Table of Contents

Student: Ghiorghe Sorana
Grupa: 30235

# Raționament și Gestionarea Sarcinilor

## Tema Proiectului: Redimensionarea imaginilor digitale (zoom and shrink)

Proiectul propune explorarea și implementarea diferitelor tehnici de scalare a imaginilor digitale grayscale pe 8 biți, atât pentru mărire, cât și pentru micșorare, indiferent de factorul de scalare. Vor fi analizate avantajele și limitările fiecărei metode, comparand tehnicile de interpolare utilizate.

## Prezentare Generală

Redimensionarea imaginilor este un proces esențial în procesarea imaginilor, permițând scalarea acestora prin mărire sau micșorare, cu scopul de a păstra cât mai multe detalii și o calitate vizuală optimă. Acest proiect își propune să exploreze și să analizeze eficiența diferitelor tehnici de interpolare utilizate pentru redimensionarea imaginilor.

Scopul principal este implementarea și compararea a trei metode distincte de interpolare, evidențiind avantajele și limitările fiecăreia. Studiul va include atât aspecte teoretice legate de interpolare, cât și implementări practice utilizând OpenCV.

## Metode implementate

În mod intuitiv, redimensionarea unei imagini presupune ajustarea numărului de pixeli, fie prin adăugarea de noi valori pentru a umple spațiile goale, fie prin eliminarea unor pixeli existenți. În acest proces, se poate pierde informație esențială, motiv pentru care este necesară o strategie de aproximare a valorilor pixelilor noi.

Din acest motiv, se utilizează metode de interpolare dezvoltate cu scopul de a estima valorile pixelilor pe baza celor existenți, menținând astfel continuitatea vizuală a imaginii. Un algoritm de interpolare trebuie să asigure un echilibru între acuratețea detaliilor și costul de procesare, astfel încât imaginea redimensionată să păstreze cât mai bine caracteristicile originale.

### 1. Interpolarea cel mai apropiat vecin (Nearest Neighbor Interpolation)

Prima metodă analizată este interpolarea celui mai apropiat vecin (*Nearest Neighbor Interpolation*), cea mai simplă abordare, unde fiecărui pixel nou i se atribuie valoarea pixelului existent cel mai apropiat. Aceasta are avantajul unei procesări rapide și este potrivită pentru imagini cu detalii simple (ex. cod de bare). Totuși, un dezavantaj major este efectul vizibil de pixelare, mai ales la mărirea imaginilor.

Student: Ghiorghe Sorana
Grupa: 30235

2. **Interpolarea biliniară (Bilinear Interpolation)**

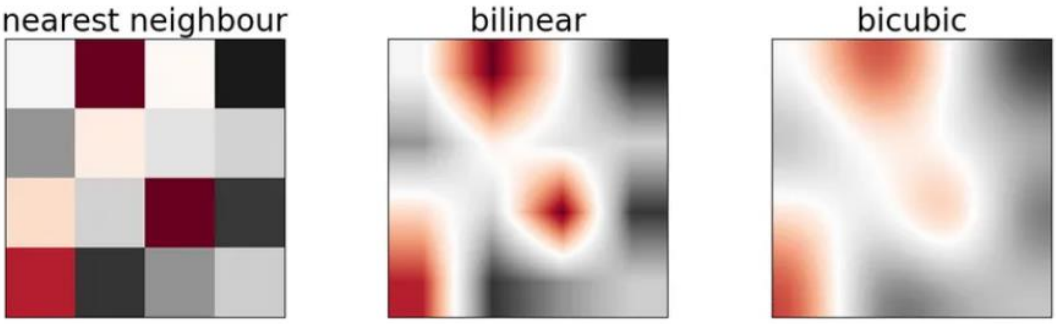
O alternativă mai avansată este interpolarea biliniară (*Bilinear Interpolation*), care calculează valoarea unui pixel necunoscut pe baza mediei ponderate a celor patru pixeli vecini (matrice 2×2). Această metodă oferă tranziții mai line și reduce efectul de pixelare, generând imagini mai fluide decât metoda precedentă. În schimb, necesită mai multe calcule, ceea ce crește timpul de procesare și poate duce la o ușoară pierdere a clarității detaliilor.

3. **Interpolarea bicubică (Bicubic Interpolation)**

Cea mai complexă metodă implementată este interpolarea bicubică (*Bicubic Interpolation*), care ia în considerare 16 pixeli vecini (matrice 4×4) pentru a calcula valoarea fiecărui pixel interpolat. Această tehnică oferă tranziții mult mai fine și rezultate mai precise comparativ cu metoda biliniară, generând imagini mai clare și mai detaliate. Cu toate acestea, complexitatea sa ridicată implică un timp de procesare mai mare.

4. **Interpolarea Lanczos (Lanczos Interpolation)**

Interpolarea Lanczos este o metodă avansată, bazată pe o funcție sinc trunchiată, care oferă o calitate superioară în special în procesul de micșorare a imaginilor. Algoritmul ia în calcul o zonă mai largă de pixeli vecini (de obicei 6×6), aplicând un kernel Lanczos pentru a păstra detaliile fine și a reduce artefactele precum aliasing-ul. Rezultatul este o imagine clară și bine definită, cu margini netede și un nivel ridicat de fidelitate vizuală. Dezavantajul principal este complexitatea computațională mai mare, ceea ce îl face mai potrivit pentru aplicații în care calitatea este prioritară față de viteza de procesare.




Prin implementarea și compararea acestor metode, proiectul propune să ofere o perspectivă clară asupra eficienței fiecăreia în diferite situații.

Student: Ghiorghe Sorana
Grupa: 30235

## Utilizare

Redimensionarea imaginilor prin interpolare este esențială în domenii care necesită procesarea și analiza imaginilor. Această metodă permite modificarea dimensiunii imaginilor fără a provoca distorsiuni majore sau pierderi semnificative de calitate, optimizând afișarea și procesarea acestora în funcție de cerințele specifice ale fiecărui context de utilizare.

Editare si procesare grafică

Utilizată în programe precum Adobe Photoshop și GIMP pentru redimensionarea imaginilor fără pierderea clarității și detaliilor vizuale.

Compresia si transmiterea imaginilor

Optimizarea imaginilor pentru încărcare rapidă pe internet și transmitere eficientă. Platformele de social media, precum Instagram, reduc dimensiunea acestora pentru viteză și compatibilitate.

Inteligenta artificiala

Interpolarea este folosită în viziunea computerizată pentru preprocesarea imaginilor înainte de utilizarea în modele de învățare automată, esențială în recunoaștere facială și conducere autonomă.

Adaptare la diferite dispozitive digitale

Scalarea imaginilor pentru afișare optimă pe monitoare, televizoare și proiectoare cu rezoluții și densități diferite de pixeli.

Domeniul medical

Utilizată în imagistica medicală (X-ray, tomografie, MRI) pentru analizarea detaliilor fine cu precizie, facilitând diagnosticarea corectă și planificarea tratamentelor

Tiparire si publicare

Ajustarea dimensiunii imaginilor pentru rezoluția specifică materialelor imprimate, asigurând calitate superioară în cărți, reviste și afișe.

Student: Ghiorghe Sorana
Grupa: 30235

## Probleme adresate

În cadrul acestui proiect, sunt analizate mai multe probleme specifice redimensionării imaginilor, fiecare având un impact direct asupra calității și eficienței procesării acestora:

Pierderea detaliilor și efectele vizuale nedorite

Micșorarea imaginilor poate duce la pierderea detaliilor fine, iar mărirea poate cauza pixelare sau neclaritate. Alegerea metodei potrivite de interpolare ajută la minimizarea acestor probleme.

Eficiența computationala

Metodele mai avansate, precum interpolarea bicubică, oferă o calitate superioară, dar necesită mai multe resurse de calcul comparativ cu cele mai simple, cum ar fi interpolarea cel mai apropiat vecin.

Scalarea neuniforma și deformarea

Ajustarea incorectă a factorilor de scalare pe axele X și Y poate duce la distorsiuni ale imaginii. Soluțiile analizate vizează menținerea proporțiilor corecte.

Student: Ghiorghe Sorana
Grupa: 30235

## Planul Proiectului

1. Studiu și cercetare

Sarcina: Explorarea conceptelor teoretice despre redimensionarea imaginilor și tehnicile de interpolare, înțelegerea principiilor matematice. Acest pas oferă o bază solidă pentru implementare.

Rezultate: Documentație inițială cu analiza metodelor.

2. Implementare

Sarcina: Dezvoltarea unui program C++ cu OpenCV pentru implementarea metodelor de interpolare în funcție de un factor de scalare, testarea algoritmilor pe imagini simple.

Rezultate: Algoritmi functionali pentru testare si comparatie.

Posibile probleme: Erori de procesare, testate pe imagini de dimensiuni și formate variate. Performanță scăzută pe imagini mari, necesitând optimizare.

3. Evaluare si comparatie

Sarcina: Compararea rezultatelor obtinute cu fiecare metodă de interpolare și evaluarea calității imaginilor.

Rezultate: Rapoarte, grafice și tabele pentru evidențierea diferentelor.

Posibile probleme: Dificultăți în măsurarea obiectivă a calității, subiectivitate în evaluare, motiv pentru care se vor folosi metrici standardizate.

4. Imbunatatirea solutiei

Sarcina: Optimizarea codului pentru performanță și eficiență a memoriei, explorarea îmbunătățirilor metodei.

Rezultate: Reducerea consumului de resurse, o versiune optimizată a codului.

Posibile probleme: Dificultăți în optimizare fără pierderi de calitate, necesitând testarea mai multor variante pentru un compromis echilibrat dintre viteză și calitate.

Student: Ghiorghe Sorana
Grupa: 30235

# Implementare

Aplicația C++ este structurată modular, cu accent pe claritate, scalabilitate și ușurință în testare. Interpolările sunt definite în fișiere separate în directorul src/, iar aplicația principală (main) utilizează OpenCV pentru manipularea imaginilor și afișarea rezultatelor.

## Funcția principală (main)

În aplicația dezvoltată, sunt utilizate două imagini: una destinată testării scalării în sus și cealaltă pentru scalarea în jos, ambele fiind încărcate în format grayscale. Motivul pentru care am ales să folosesc 2 imagini separate este pentru a observa diferitele efecte in urma aplicarii algoritmilor. Am urmarit efectele interpolarii atat pe imagini binare, pentru a observa efectul de ringing in metoda bicubica (si eficienta metodei nearest neighbour in salvarea valorilor efective ale pixelilor). De asemenea, am urmarit efectele interpolarii si pe o imagine mica, cu putini pixeli, pentru a observa felul in care diferiti algoritmi aproximeaza valorile pixelilor lipsa. Efectul de scalare in jos am vrut sa il observ pe o imagine mai mare, pentru a ma asigura ca detaliile predominante sunt pastrate.

In cadrul acestei functii

- Se definesc parametrii de scalare in sus si in jos, setati la valoarea 2.0 si respectiv 0.4. Valorile sub 1.0 reprezinta scalarea in jos, iar cele peste 1.0 scalarea in sus. Acestia pot fi modificati pentru a vizualiza efectele mai drastice in procesare.

```
double scaleUpX = 2.0, scaleUpY = 2.0;
double scaleDownX = 0.4, scaleDownY = 0.4;
```

- Se aplica metoda de scalare in sus cu urmatoarele metode de interpolare: Nearest Neighbour, Bilinear, Bicubic. Interpolarea Bicubică are un parametru a, care modifică functia din kernel ul cubic. Am implementat astfel interpolarea bicubică pentru a observa influenta acestui parametru in scalarea fiecărei imagini, deoarece prin parametru implicit limitam rezultatele scalarii si nu se observau efectele negative atat de clar. Parametrul a fost ales cu o valoare foarte mare si foarte mică în mod intentionat, pentru a face efectele vizibile cu ochiul liber.

```
Mat bicubic_CatmullRom = bicubicCustom(scaleUpImg, scaleUpX, scaleUpY, a: -0.5f);
Mat bicubic_BSpline = bicubicCustom(scaleUpImg, scaleUpX, scaleUpY, a: 0.0f);
Mat bicubic_Sharper = bicubicCustom(scaleUpImg, scaleUpX, scaleUpY, a: -5.0f);
Mat bicubic_Smoother = bicubicCustom(scaleUpImg, scaleUpX, scaleUpY, a: 5.0f);
```

- Se aplica metoda de scalare in jos cu interpolarea Lanczos, pentru a pastra structura originala. Interpolarea Lanczos poate fi aplicată si pe scalarea în sus.

Student: Ghiorghe Sorana
Grupa: 30235

## Metode de interpolare implementate

Metodele de interpolare sunt implementate intr-un fisier .cpp separat, numit interpolation.cpp, cu

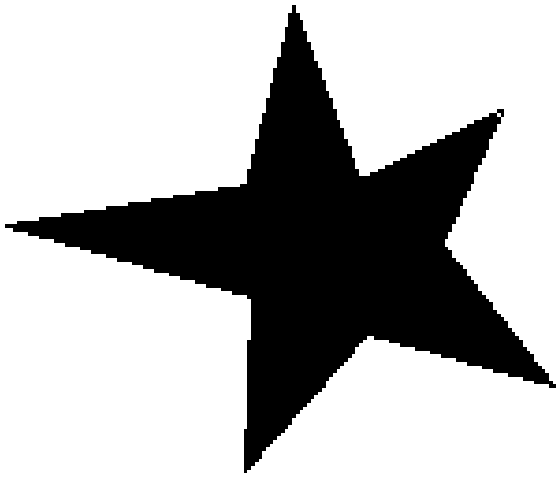
```
Mat nearestNeighbor(const Mat& input, double scaleX, double scaleY);
Mat bilinear(const Mat& input, double scaleX, double scaleY);
Mat bicubicCustom(const Mat& input, double scaleX, double scaleY, float a);
Mat lanczos(const Mat& input, double scaleX, double scaleY);
```

un header prin care facem referinta la toate metodele.

In intermediul proiectului, exista si functii auxiliare pentru gestionarea corecta a valorilor si asigurarea unui comportament robust si asteptat.

- clamp() care limitează valorile în intervalul valid al pixelilor.
- getPixelValue() care accesează în siguranță valoarea unui pixel, evitând ieșirea din matrice
- bicubicKernel() care reprezinta functii de gradul 3 pentru interpolarea bicubica.
- lanczosKernel() si sinc() care reprezinta functiile matematice pentru reprezentarea kernelului lanczos.
- getImage() care reprezinta o functie in main pentru a simplifica procesul de citire a unei imagini si afisarea ei cu imread si imshow.

**Imaginea originala:**



Student: Ghiorghe Sorana
Grupa: 30235

**Interpolarea Nearest Neighbour**

Implementarea constă în atribuirea fiecărui pixel din imaginea redimensionată valorii pixelului cel mai apropiat din imaginea originală. Algoritmul parcurge fiecare coordonată calculată prin raportul de scalare și mapează direct pe coordonatele cele mai apropiate întregi. Avantajul este simplitatea codului și viteza mare de execuție, fiind implementat prin conversie directă a pozițiilor în coordonate întregi. Nu necesită kerneluri sau calcule adiționale.

```
Mat nearestNeighbor(const Mat& input, double scaleX, double
scaleY) {
   // dimensiunea imaginii output
   int newW = input.cols * scaleX;
   int newH = input.rows * scaleY;
   Mat output(newH, newW, CV_8UC1); // grayscale

   //loop pe imaginea output
   for (int y = 0; y < newH; ++y) {
      for (int x = 0; x < newW; ++x) {
         int srcX = static_cast<int>(x / scaleX);
         int srcY = static_cast<int>(y / scaleY);
         //static_cast<int> trunchiaza rezultatul la cel mai apropiat
         // pixel (sus-stanga)
         //x / scaleX asociere intre pixel original si pixel nou
         output.at<uchar>(y, x) = input.at<uchar>(srcY, srcX);
      }
   }
   return output;
}
```

Efectul Nearest Neighbour

$$f(x) = \begin{cases} f(\lfloor x \rfloor) & \text{for } x - \lfloor x \rfloor < 1/2 \\ f(\lfloor x \rfloor + 1) & \text{otherwise} \end{cases}$$

Student: Ghiorghe Sorana
Grupa: 30235

Funcția **nearestNeighbor** implementează metoda de interpolare cel mai apropiat vecin, una dintre cele mai simple și rapide tehnici de redimensionare a imaginilor. Principiul fundamental este următorul: pentru fiecare pixel din imaginea de ieșire, se identifică pixelul corespunzător din imaginea originală printr-o proiecție inversă și se copiază direct valoarea acelui pixel.

Etapele sunt:

1. Se calculează dimensiunea imaginii de ieșire pornind de la factorii de scalare
2. Pentru fiecare pixel din imaginea de ieșire:
    - Se determină coordonatele sursă (srcX, srcY) prin inversarea scalei.
    - Coordonatele sunt trunchiate la cel mai apropiat întreg, echivalent cu alegerea pixelului sursă cel mai apropiat (in general sus-stânga).
    - Se copiază valoarea pixelului respectiv din imaginea originală în pixelul de ieșire.

Este o metodă fără interpolare propriu-zisă – se realizează doar o asociere discretă între poziții, fără a evalua valori intermediare.

Aceasta metodă a fost implementată în cadrul proiectului drept un punct de referință, fiind cea mai simplă metodă, permite evaluarea comparativă a calității și performanței celorlalte metode (bilineară, bicubică, Lanczos) si este singura metodă care pastrează cu exactitate datele imaginii.

Inițial am testat rotunjirea coordonatelor cu round() în loc de static_cast<int>, însă aceasta a generat artefacte vizibile la margini din cauza suprautilizării unor pixeli și omiterii altora, în timp ce trunchierea a oferit rezultate mai stabile, mai ales la scale fracționare.

Student: Ghiorghe Sorana
Grupa: 30235

**Interpolarea Biliniara**

Interpolarea biliniară este implementată prin extragerea celor patru pixeli vecini (matrice 2x2) și calcularea unei medii ponderate în două trepte: mai întâi pe linie, apoi pe coloană. Algoritmul este eficient, necesitând doar operații liniare și este sensibil la ordinea în care se fac interpolările. În cod, am evitat erori de acces folosind `getPixelValue()` și am aplicat interpolarea doar pentru canale de 8 biți.

```
Mat bilinear(const Mat& input, double scaleX, double scaleY) {
  // dimensiunea imaginii output
  int newW = input.cols * scaleX;
  int newH = input.rows * scaleY;
  Mat output(newH, newW, CV_8UC1); // grayscale

  //loop pe imaginea output
  for (int y = 0; y < newH; ++y) {
    for (int x = 0; x < newW; ++x) {
      // calc coordonatele din imaginea input
      float gx = x / scaleX;
      float gy = y / scaleY;
      // floor = cea mai mare val intreaga
      int x0 = floor(gx);
      int y0 = floor(gy);
      // vecin right
      int x1 = x0 + 1;
      // vecin bottom
      int y1 = y0 + 1;
      //diferenta fractionala pt interpolare
      float dx = gx - x0; //diferenta orizontala
      float dy = gy - y0; //diferenta verticala
      // 4 surrounding pixels
      uchar p00 = getPixel(input, x0, y0); // top-left
      uchar p10 = getPixel(input, x1, y0); // top-right
      uchar p01 = getPixel(input, x0, y1); // bottom-left
      uchar p11 = getPixel(input, x1, y1); // bottom-right
      //interpolare pe baza distantei (formula)
      float value =
        (1 - dx) * (1 - dy) * p00 +
        dx * (1 - dy) * p10 +
        (1 - dx) * dy * p01 +
        dx * dy * p11;
      output.at<uchar>(y, x) = static_cast<uchar>(value);
    }
  }
  return output;
}
```

Efectul Bilinear Interpolation

Funcția **bilinear** implementează interpolarea bilineară, o tehnică eficientă și des utilizată pentru redimensionarea imaginilor. Spre deosebire de interpolarea bicubică, aceasta se bazează pe o vecinii de 2×2 pixeli și aplică o interpolare liniară în două direcții (orizontală și verticală).

$$f_{y1} = f_{11} + \frac{f_{21} - f_{11}}{x_2 - x_1}(x - x_1)$$

$$f_{y2} = f_{12} + \frac{f_{22} - f_{12}}{x_2 - x_1}(x - x_1)$$

$$f(x, y) = f_{y1} + \frac{f_{y2} - f_{y1}}{y_2 - y_1}(y - y_1)$$

$$\begin{aligned} x_1 &= \lfloor x \rfloor & f_{11} &\equiv f(x_1, y_1) \\ x_2 &= \lfloor x \rfloor + 1 & f_{12} &\equiv f(x_1, y_2) \\ y_1 &= \lfloor y \rfloor & f_{21} &\equiv f(x_2, y_1) \\ y_2 &= \lfloor y \rfloor + 1 & f_{22} &\equiv f(x_2, y_2) \end{aligned}$$

Student: Ghiorghe Sorana
Grupa: 30235

Pentru fiecare pixel (x, y) din imaginea de ieșire:

1. Se calculează poziția corespunzătoare (gx, gy) în imaginea originală, ținând cont de factorii de scalare.
2. Se identifică cei patru pixeli care formează un pătrat de 2×2: (x0, y0), (x1, y0), (x0, y1), (x1, y1)
3. Se calculează diferențele fracționare dx și dy, care indică poziția relativă a punctului în interiorul pătratului.
4. Se face o interpolare liniară dublă (bilineară) între cei patru pixeli, rezultând valoarea estimată a pixelului la poziția (x, y) în imaginea redimensionată.

Această metodă oferă o calitate vizuală mai bună decât interpolarea nearest-neighbor, dar cu un cost computațional mult mai mic decât interpolarea bicubică.

Interpolarea bilineară a fost implementată datorită eficienței sale computaționale, având o complexitate redusă care o face potrivită pentru aplicații rapide fără o pierdere semnificativă de calitate. Simplitatea algoritmului îl face ușor de înțeles și adaptat, fiind ideal pentru scopuri educaționale sau aplicații embedded. Metoda oferă un compromis excelent între calitate și performanță, depășind semnificativ metoda nearest-neighbor, cu un cost computațional minim.

Funcția permite scalare flexibilă pe axele X și Y prin parametrii scaleX și scaleY, oferind o interpolare precisă prin media ponderată a celor patru pixeli vecini. Accesul la valorile sursă este gestionat în mod robust prin funcția getPixel, asigurând stabilitate la margini. Datorită performanței ridicate, metoda este potrivită și pentru procesare în timp real sau pe sisteme cu resurse limitate.

Student: Ghiorghe Sorana
Grupa: 30235

## Interpolarea Bicubica

Pentru interpolarea bicubică, am folosit o matrice 4x4 de pixeli în jurul fiecărei poziții target, aplicând un kernel cubic personalizabil.

```
float cubicKernel(float x, float a) {
   x = abs(x); // non negative vals
   if (x <= 1)
      // distanta mica
      return (a + 2) * x * x * x - (a + 3) * x * x + 1;
   else if (x < 2)
      return a * x * x * x - 5 * a * x * x + 8 * a * x - 4 * a;
   return 0.0f; //distanta mare, nu influenteaza pixelul
}
```

$$f(x; -1 \le a < 0) = \begin{cases} (a+2)|x|^3 - (a+3)x^2 + 1 & \text{for } 0 \le |x| \le 1 \\ a|x|^3 - 5ax^2 + 8a|x| - 4a & \text{for } 1 < |x| \le 2 \\ 0 & \text{otherwise} \end{cases}$$

Am definit funcția cubicKernel(float x, float a) cu parametrul a reglabil pentru a putea experimenta cu rezultatele interpolării.

O valoare standard pentru a este -0.5, astfel formand spline-ul Catmull-Rom.

Valorile mai mici ale parametrului a ofera un efect mai puternic de blur in jurul linilor (asadar se pierde informatia din imaginea originala), iar valorile mai mari ale parametrului ofera un efect de ascutime, dar care duce inspre efectul de ringing.

Inițial am încercat preprocesarea kernelului și memorarea într-un tabel (lookup table) pentru a evita recalculările. Totuși, soluția consuma prea multă memorie pentru imagini mari, nu aducea un câștig semnificativ de performanță și limita flexibilitatea, deoarece parametrul a nu putea fi ajustat fără recalcularea tabelului. Am optat astfel pentru calculul direct al kernelului cubic în execuție, menținând un echilibru între performanță, claritate și adaptabilitate.

Efectul Bicubic Interpolation cu a = -0.5f (Catmull-Rom)

Efectul Bicubic Interpolation cu a = -5.0f

Efectul Bicubic Interpolation cu a = 5.0f

Parametrul a fost ales in aceste extreme in mod intentionat, pentru a putea observa cu ochiul liber influenta acestuia in functia matematica.

Student: Ghiorghe Sorana
Grupa: 30235

```
Mat bicubicCustom(const Mat& input, double scaleX,
double scaleY, float a = -0.5f) {
   // dimensiunea imaginii output
   int newW = input.cols * scaleX;
   int newH = input.rows * scaleY;
   Mat output(newH, newW, CV_8UC1); // grayscale
   for (int y = 0; y < newH; ++y) {
      for (int x = 0; x < newW; ++x) {
         // calc coordonatele din imaginea input
         float gx = x / scaleX;
         float gy = y / scaleY;
         // floor = cea mai mare val intreaga
         int x0 = floor(gx);
         int y0 = floor(gy);
         //diferenta fractionala pt interpolare
         float dx = gx - x0; //diferenta orizontala
         float dy = gy - y0; //diferenta verticala
         //init sumele
         // weight = mai mare = mai aproape, influenteaza
mai mult pixelul care trebuie aproximat
         float sum = 0;
         float weightSum = 0;
         // 4x4 neighbourhood
         for (int m = -1; m <= 2; ++m) {
            for (int n = -1; n <= 2; ++n) {
               //folosim kerneul cubic parametrizat pentru x
si y
               float w = cubicKernel(m - dx, a) *
cubicKernel(dy - n, a);
               sum += getPixel(input, x0 + m, y0 + n) * w;
//adaugam pixelul in suma
               weightSum += w; // ptr a impartii la final
            }
         }
         // normalizam dupa weight - ul total si asignam
valoarea pixelului output
         output.at<uchar>(y, x) =
saturate_cast<uchar>(sum / weightSum);
      }
   }
   return output;
}
```

Funcția **bicubicCustom** implementează o versiune personalizată a algoritmului de interpolare bicubică, folosit pentru redimensionare. Algoritmul funcționează aplicând un kernel cubic parametrizat (controlat de parametrul a) pentru a estima valorile pixelilor în imaginea de ieșire, pornind de la un vecinii 4×4 al pixelilor sursă.

Pașii principali ai algoritmului sunt:

- Se calculează dimensiunea noii imagini în funcție de factorii scaleX și scaleY.

- Pentru fiecare pixel din imaginea de ieșire:

1. Se determină poziția sa corespunzătoare în imaginea originală (gx, gy).

2. Se identifică vecinătatea de 4×4 pixeli în jurul poziției originale.

3. Se aplică funcția cubicKernel pentru a calcula ponderi (weights) pentru fiecare pixel vecin.

4. Se face media ponderată a pixelilor vecini, rezultând valoarea finală a pixelului.

Implementarea actuală oferă control asupra calității imaginii prin parametrul a, care ajustează forma kernelului cubic pentru diverse experimente. Este modulară și extensibilă, separând clar kernelul de logica interpolării, facilitând adaptarea pentru imagini color sau integrarea în alte module.

Student: Ghiorghe Sorana
Grupa: 30235

## Interpolarea Lanczos

Implică folosirea unei funcții sinc trunchiate in kernel, definită în lanczosKernel() cu suport de 3 (matrice 6x6). Este folosită în special pentru scalare în jos, pentru a minimiza aliasing-ul. Interpolarea Lanczos este mai costisitoare din punct de vedere al performanței din cauza dimensiunii mai mari a kernelului. Implementarea a fost făcută calculând valorile pe direcțiile X și Y separat, pentru a face procesul mai eficient. În plus, accesul la pixeli a fost limitat doar la zona apropiată unui pixel, în funcție de dimensiunea kernelului, pentru a îmbunătăți viteza.

```
float sinc(float x) {
   if (x == 0) return 1.0;
   x *= CV_PI;
   return sin(x) / x;
}

float lanczosKernel(float x, int n = 3) {
   x = abs(x); // val pozitive
   if (x <= n) return sinc(x) * sinc(x / n); //function
   return 0; //fara contributii de la kernel
}
```

$$\mathrm{L}(x; n > 0) = \begin{cases} \mathrm{sinc}(x) \cdot \mathrm{sinc}(x/n) & \text{for } |x| \leq n \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{sinc}(x) = \begin{cases} 1 & \text{for } x = 0 \\ \dfrac{\sin(\pi x)}{\pi x} & \text{otherwise} \end{cases}$$

La fel ca în cazul interpolării bicubice, am încercat initial o versiune care calcula și memora valorile kernelului pentru a îmbunătăți performanța, însă dimensiunea mare a tabelului (proporțională cu $a^2$) a dus la un consum crescut de memorie și a limitat flexibilitatea, deoarece nu se putea modifica dinamic parametrul a. Astfel, s-a optat pentru evaluarea directă a kernelului la fiecare punct, folosind o implementare optimizată a funcției lanczosKernel.

Efectul micsorării Lanczos imaginii la o scala de 0.4 cu a

Student: Ghiorghe Sorana
Grupa: 30235

```
Mat lanczos(const Mat& input, double scaleX, double
scaleY, int a) {
   // dimensiunea imaginii output
   int newW = input.cols * scaleX;
   int newH = input.rows * scaleY;
   Mat output(newH, newW, CV_8UC1); // grayscale

   for (int y = 0; y < newH; ++y) {
      for (int x = 0; x < newW; ++x) {
         // calc coordonatele din imaginea input
         float gx = x / scaleX;
         float gy = y / scaleY;
         // floor = cea mai mare val intreaga
         int x0 = floor(gx);
         int y0 = floor(gy);



         //init sumele
         // weight = mai mare = mai aproape,
influenteaza mai mult pixelul care trebuie aproximat
         float sum = 0;
         float weightSum = 0;

         for (int m = -a + 1; m <= a; ++m) {
            for (int n = -a + 1; n <= a; ++n) {
               float dx = gx - (x0 + m); // diferenta
orizontala
               float dy = gy - (y0 + n); // diferenta verticala
               float w = lanczosKernel(dx, a) *
lanczosKernel(dy, a);
               sum += getPixel(input, x0 + m, y0 + n) * w;
//cu pondere
               weightSum += w; // ptr a impartii la final
            }
         }

         output.at<uchar>(y, x) =
saturate_cast<uchar>(sum / weightSum);
      }
   }

   return output;
}
```

Funcția **lanczos** implementează interpolarea Lanczos, o metodă precisă bazată pe o funcție sinc trunchiată, cunoscută pentru redimensionarea imaginii si păstrarea clarității contururilor, cu minimizarea artefactelor vizuale(aliasingul și efectul de blur)
Interpolarea Lanczos funcționează evaluând o funcție sinc modificată în jurul poziției sursă corespunzătoare fiecărui pixel din imaginea scalată. Parametrul a (de obicei între 2 și 4) controlează câți pixeli vecini sunt luați în calcul pentru fiecare estimare.
Pașii principali ai algoritmului sunt:

- Se determină dimensiunea imaginii de ieșire.
- Pentru fiecare pixel din imaginea de ieșire:
    1. Se calculează poziția sursă (gx, gy) în imaginea originală.
    2. Se identifică o vecinătate pătrată de (2a)×(2a) pixeli în jurul punctului sursă.
    3. Se evaluează kernelul în fiecare direcție (X și Y), iar valoarea pixelului este calculată ca media ponderată a pixelilor vecini.
    4. Ponderile sunt date de produsul dintre valorile kernelului în direcțiile orizontală și verticală.

Această trunchiere reduce efectul negativ al sinc-ului infinit, păstrând însă acuratețea spectrală ridicată.
Interpolarea Lanczos a fost aleasă datorită calității excelente a rezultatelor, oferind o reconstrucție detaliată cu tranziții line și margini clare.
Parametrul a permite controlul între precizie și viteză, iar metoda are o bază solidă din punct de vedere teoretic, fiind stabilă atât numeric, cât și în contextul analizei semnalelor.

Student: Ghiorghe Sorana
Grupa: 30235

# Testare

Pentru evaluarea performanței fiecărui algoritm de interpolare implementat, am realizat o serie de teste comparative între metodele proprii și cele oferite de biblioteca OpenCV. Testarea a fost efectuată pe trei imagini grayscale, folosind trei factori de scalare diferiți (zoom in, zoom out și scalare neuniformă), cu scopul de a obține o imagine clară asupra acurateței și eficienței fiecărei metode.

Pentru fiecare combinație de imagine, scalare și algoritm, au fost calculate următoarele metrici:

- **Timp de execuție (ms)** măsoară performanța algoritmului din punct de vedere al vitezei. Este esențială în contexte în care timpul de procesare contează (ex: aplicații în timp real).
- **PSNR (Peak Signal-to-Noise Ratio)** evaluează diferențele globale dintre imaginea inițială și cea procesată. Valori mai mari indică o fidelitate mai bună față de original.
- **SSIM (Structural Similarity Index)** măsoară asemănarea structurală dintre imagini, ținând cont de factori perceptuali precum luminozitate, contrast și textură. Este considerată o metrică mai apropiată de percepția umană.
- **RMSE (Root Mean Square Error)** oferă media pătratică a erorilor pe pixeli, reflectând deviațiile locale dintre imaginile comparate.

În cazul algoritmilor **Bicubic** și **Lanczos**, am testat și comportamentul acestora la modificarea parametrului a, care influențează forma și comportamentul funcției de interpolare. Astfel, am putut observa modul în care schimbarea parametrilor afectează calitatea rezultatelor atât în mod numeric cât și vizual.

Toate rezultatele sunt salvate atât în format txt pentru afișare detaliată în consolă, cât și în format csv pentru o analiză ulterioară mai ușoară.

Testele au fost efectuate pentru 3 factori de scalare (2x2, 0.5x0.5 si 1.5x0.75).

Student: Ghiorghe Sorana
Grupa: 30235

| Image | Scale | Algorithm | Time (ms) | PSNR | SSIM | RMSE |
|---|---|---|---|---|---|---|
| balloons | x1.5 | Nearest | 11 | 29.9018 | 0.95251 | 8.15545 |
| balloons | x1.5 | Bilinear | 41 | 32.8503 | 0.97245 | 5.808 |
| balloons | x1.5 | Bicubic (a=-0.5) | 373 | 33.8837 | 0.97844 | 5.15649 |
| balloons | x1.5 | Bicubic (a=-0.7) | 367 | 34.0983 | 0.97965 | 5.03068 |
| balloons | x1.5 | Bicubic (a=-1.0) | 380 | 34.1989 | 0.98038 | 4.97272 |
| balloons | x1.5 | Lanczos (a=2) | 823 | 33.862 | 0.97838 | 5.1694 |
| balloons | x1.5 | Lanczos (a=3) | 1849 | 34.2106 | 0.9802 | 4.96603 |
| balloons | x1.5 | Lanczos (a=4) | 3777 | 34.1689 | 0.98007 | 4.98996 |
| balloons | x1.5 | OpenCV Nearest | 0 | 29.9018 | 0.95251 | 8.15545 |
| balloons | x1.5 | OpenCV Bilinear | 2 | 33.9652 | 0.97662 | 5.10836 |
| balloons | x1.5 | OpenCV Bicubic | 0 | 36.4554 | 0.9858 | 3.83504 |
| balloons | x1.5 | OpenCV Lanczos | 4 | 36.6678 | 0.98642 | 3.74238 |

Acesta este un exemplu de output pentru testarea fiecarui algoritm pe o scalare de 1.5x0.75 neuniformă. In acest caz se poate identifica mai usor influenta parametriilor in interpolarea bicubica si lanczos.

La bicubica, am folosit 3 parametrii reali ( -0.5, -0.7 si -1.0 comparativ cu parametrii drastici implementati initial, deoarece am considerat ca numeric se vor observa diferentele mult mai bine decat cum sar fi observat doar cu ochiul liber, insa concluzia la care am ajuns este ca parametrul in sine nu influenteaza atat de tare rezultatele. Un parametru mai mic la Bicubic ofera rezultate mai sharp, insa in cresterea acestuia am fi provocat efectul de ringing nedorit.

Student: Ghiorghe Sorana
Grupa: 30235

## Evaluare Metoda Nearest Neighbour

Metoda Nearest Neighbor este singura care păstrează complet detaliile originale ale imaginii, astfel încât la o scalare uniformă (de exemplu, upscaling x2) rezultatul este identic cu imaginea inițială. Acest lucru se datorează faptului că nu există pierderi de detalii, iar reconstruirea imaginii se realizează foarte simplu.

Pentru ceilalți factori de scalare, se observă că valorile PSNR, SSIM și RMSE obținute prin această metodă sunt comparabile cu cele ale implementării built-in din OpenCV, ceea ce confirmă corectitudinea și consecvența rezultatelor.

Deși Nearest Neighbour are performanțe slabe din punct de vedere calitativ în comparație cu celelalte metode de interpolare, ea compensează printr-un timp de execuție extrem de redus, fiind astfel potrivită pentru situațiile în care viteza de procesare este prioritară.

Rezultate medie nearest neighbour:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 41.44ms | 23.98 | 0.8847 | 18.5766 |

Rezultate medie nearest neighbour OpenCV:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 3.11ms | 23.82 | 0.884 | 18.76 |

Student: Ghiorghe Sorana
Grupa: 30235

# Evaluare Metoda Bilineara

Metoda **Bilineara** realizează o interpolare liniară pe două direcții, oferind un echilibru bun între calitatea imaginii și timpul de execuție. Comparativ cu metoda Nearest Neighbor, aceasta oferă rezultate vizibil mai bune din punct de vedere al metricilor PSNR, SSIM și RMSE, traduse printr-o calitate vizuală superioară și o mai bună păstrare a detaliilor.

Timpul de execuție pentru această metodă este mai ridicat decât al Nearest Neighbor, dar rămâne rezonabil pentru majoritatea aplicațiilor. Valorile medii obținute în urma testelor arată că metoda Bilinear oferă o calitate consistentă și stabilă a imaginii scalate, cu un PSNR mediu de aproximativ 26.34, un SSIM mediu de 0.925 și un RMSE mediu de 13.77.

Această metodă este recomandată atunci când este necesară o calitate mai bună a imaginii, fără a compromite prea mult performanța procesării.

Rezultate medie metoda bilinear:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 121.22ms | 26.34 | 0.925 | 13.77 |

Rezultate medie metoda bilinear OpenCV:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 3.33ms | 27.43 | 0.943 | 11.73 |

Student: Ghiorghe Sorana
Grupa: 30235

# Evaluare metoda Bicubica

Metoda **Bicubică** este o interpolare de ordin superior care folosește funcții cubice pentru a calcula valorile pixelilor noi, rezultând o imagine mult mai netedă și cu detalii mai bine păstrate comparativ cu metodele Nearest Neighbor sau Bilinear.

Un aspect important al acestei metode este parametrul *a*, care controlează forma și netezimea kernel-ului cubic utilizat. Valorile uzuale pentru *a* sunt în intervalul [-1, 0], iar valoarea *a = -0.5* (cunoscuta ca interpolare Catmull-Rom) oferă un echilibru bun între netezime și păstrarea detaliilor. Modificarea parametrului *a* influențează claritatea imaginii finale, dar diferențele în metrici precum PSNR sau SSIM sunt de obicei subtile.

În ceea ce privește performanța, metoda bicubică are un timp de execuție mult mai ridicat decât metodele anterioare (în medie peste 1000 ms în testele efectuate), însă calitatea imaginii scalate se îmbunătățește vizibil, cu un PSNR mediu de aproximativ 26.88, un SSIM mediu de 0.925 și un RMSE mediu de 13.07.

Această metodă este recomandată în special atunci când calitatea imaginii este prioritară, iar timpul de procesare poate fi acceptat ca fiind mai mare

Rezultate medie pentru interpolarea bicubica, cu parametru a=-0.5.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 1007.89ms | 26.88 | 0.9253 | 13.07 |

Rezultate medie pentru interpolarea bicubica, cu parametru a=-0.5 in OpenCV.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 0.33ms | 29.72 | 0.944 | 11.73 |

Student: Ghiorghe Sorana
Grupa: 30235

# Evaluare metoda Lanczos

Metoda **Lanczos** este o tehnică de interpolare avansată, bazată pe o funcție sinc trunchiată, care oferă o calitate superioară a imaginii scalate, păstrând detalii fine și contururi clare. Parametrul *a* reprezintă raza de truncare a nucleului Lanczos, influențând compromisul între claritatea imaginii și complexitatea calculului. Valorile obișnuite pentru *a* sunt între 2 și 4, cu *a = 2* fiind o alegere frecventă pentru echilibru optim.

Analiza rezultatelor arată că metoda Lanczos are un timp de execuție semnificativ mai mare față de metodele precedente, cu o medie de aproximativ 1600 ms, dar oferă o calitate ridicată a imaginii: un PSNR mediu de 26.74, un SSIM mediu de 0.925 și un RMSE mediu de 13.01.

Această metodă este preferată atunci când se dorește un rezultat foarte precis și detaliat, iar timpul de procesare nu este o constrângere majoră.

Rezultate medie pentru interpolarea lanczos, cu parametru a=2.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 1603.7ms | 26.74 | 0.9252 | 13.01 |

Rezultate medie pentru interpolarea lanczos, cu parametru a=2 in OpenCV.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 5.22ms | 30.99 | 0.9437 | 9.67 |

Student: Ghiorghe Sorana
Grupa: 30235

## Concluzii

Metodele de interpolare implementate de la zero au oferit rezultate bune, însă metodele built-in din OpenCV prezintă un avantaj clar în ceea ce privește timpul de execuție și calitatea imaginii scalate, pe majoritatea măsurătorilor. În special, Lanczos-ul built-in obține un PSNR mediu mai ridicat (aproximativ 31) și un RMSE mai mic (9.67) comparativ cu implementarea personalizată (PSNR mediu 26.74 și RMSE 13.01), indicând o calitate vizuală superioară.

De asemenea, timpul de execuție al implementării proprii este mult mai mare în timp ce varianta OpenCV este de ordinul câtorva milisecunde (medie 5.22 ms pentru Lanczos). Acest lucru sugerează că optimizările și algoritmii folosiți în bibliotecă sunt mult mai eficienți din punct de vedere computațional.

Student: Ghiorghe Sorana
Grupa: 30235

# Îmbunătățiri

În urma etapei de testare și analiză a performanței, am identificat două direcții majore de îmbunătățire a proiectului: extinderea suportului pentru imagini color și optimizarea performanței prin paralelizare. Aceste schimbări contribuie atât la scalabilitatea funcțională, cât și la eficiența procesării.

## Suport pentru imagini color

Inițial, implementările algoritmilor suportau doar imagini grayscale (CV_8UC1). Totuși, majoritatea aplicațiilor reale (editare foto, vizualizare, rețele neurale etc.) operează cu imagini color (BGR, CV_8UC3). Fără această extindere, algoritmii de interpolare aveau o aplicabilitate limitată.

Fiecare metodă de interpolare (Nearest Neighbor, Bilinear, Bicubic, Lanczos) a fost adaptată pentru a verifica numărul de canale din imagine. Dacă imaginea are 3 canale, se aplică interpolarea individual pe fiecare canal B, G și R. Funcția getColorPixel() a fost introdusă pentru a gestiona elegant accesul la pixeli și pentru pixeli color, care au nevoie de o structura de Vec3.

Algoritmii au fost adaptati cu o această structură:

```
if (input.channels() == 1) {
   // procesare grayscale
} else {
   // procesare color pe fiecare canal
}
```

Student: Ghiorghe Sorana
Grupa: 30235

Imaginea originală:

Scalare Nearest Neighbour:

Scalare Bilinear Interpolation:

Scalare Bicubic Interpolation:

Scalare Lanczos Interpolation:

Student: Ghiorghe Sorana
Grupa: 30235

## Paralelizare cu cv::parallel_for_

Timpul de execuție, în special pentru metodele Bicubic și Lanczos, era semnificativ mai mare comparativ cu funcțiile native OpenCV. Pentru imagini mari, acest lucru devenea o problemă majoră. Folosind procesoare moderne cu multiple nuclee, putem reduce timpul total prin procesare paralelă.

Am înlocuit bucla exterioară pe rânduri, permițând executarea simultană a mai multor linii de imagine. Această modificare a fost aplicată în toate cele 4 metode și are un impact semnificativ mai ales la scălări mari sau în batch-uri de imagini.

```
for (int y = 0; y < newH; ++y) {
   for (int x = 0; x < newW; ++x) {
      // ...
   }
}
```

```
parallel_for_(Range(0, newH), [&](const Range& range) {
   for (int y = range.start; y < range.end; ++y) {
      for (int x = 0; x < newW; ++x) {
         // ...
      }
   }
});
```

Am rulat din nou testele pe algoritmii modificați și diferențele au fost majore în timpul de executiei, cu o imbunătățire de până la 87% în cazul interpolării bicubice.

Student: Ghiorghe Sorana
Grupa: 30235

# Îmbunătățire Metoda Nearest Neighbour

Comparativ cu implementarea inițială, versiunea îmbunătățită a algoritmului Nearest Neighbour prezintă un salt semnificativ de performanță, reducând timpul de execuție de la 41.44 ms la 9.1 ms (o îmbunătățire de aproximativ 78%), fără a compromite semnificativ calitatea imaginii.

Rezultate medie nearest neighbour înainte:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 41.44ms | 23.98 | 0.8847 | 18.5766 |

Rezultate medie nearest neighbour îmbunătățită:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 9.1ms | 23.65 | 0.8847 | 18.2433 |

Rezultate medie nearest neighbour OpenCV:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 3.11ms | 23.82 | 0.884 | 18.76 |

Student: Ghiorghe Sorana
Grupa: 30235

# Îmbunătățire Metoda Bilineara

Versiunea îmbunătățită a metodei Bilineare aduce o optimizare semnificativă a timpului de execuție, de la 121.22 ms la 17.33 ms, ceea ce înseamnă o reducere de aproximativ 85.7%, cu o păstrare rezonabilă a calității reconstrucției.

Rezultate medie metoda bilinear:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 121.22ms | 26.34 | 0.925 | 13.77 |

Rezultate medie metoda bilinear îmbunătățită:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 17.33ms | 26.26 | 0.920 | 14.15 |

Rezultate medie metoda bilinear OpenCV:

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 3.33ms | 27.43 | 0.943 | 11.73 |

Student: Ghiorghe Sorana
Grupa: 30235

## Imbunătățire metoda Bicubica

Versiunea îmbunătățită a metodei bicubice cu parametrul a=−0.5 aduce o reducere semnificativă a timpului de execuție, de la 1007.89 ms la 126.11 ms (aproximativ 87.5% îmbunătățire), însă în detrimentul unei ușoare scăderi a calității imaginii rezultate.

Rezultate medie pentru interpolarea bicubica, cu parametru a=-0.5.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 1007.89ms | 26.88 | 0.9253 | 13.07 |

Rezultate medie pentru interpolarea bicubica îmbunătățită, cu parametru a=-0.5.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 126.11ms | 26.49 | 0.9236 | 13.90 |

Rezultate medie pentru interpolarea bicubica, cu parametru a=-0.5 in OpenCV.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 0.33ms | 29.72 | 0.944 | 11.73 |

Student: Ghiorghe Sorana
Grupa: 30235

## Imbunătățire metoda Lanczos

Versiunea îmbunătățită a metodei Lanczos cu parametrul a=2 reduce semnificativ timpul de execuție, de la 1603.7 ms la 315.77 ms, reprezentând o îmbunătățire de aproximativ 80.3%. Calitatea imaginii rezultate este usor mai scăzută decât în prima iteratie.

Rezultate medie pentru interpolarea lanczos, cu parametru a=2.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 1603.7ms | 26.74 | 0.9252 | 13.01 |

Rezultate medie pentru interpolarea lanczos îmbunătățită, cu parametru a=2.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 315.77ms | 26.49 | 0.9235 | 13.91 |

Rezultate medie pentru interpolarea lanczos, cu parametru a=2 in OpenCV.

| Timp Executie | PSNR | SSIM | RMSE |
|---|---|---|---|
| 5.22ms | 30.99 | 0.9437 | 9.67 |

Student: Ghiorghe Sorana
Grupa: 30235

# Bibliografie

**Formule pentru interpolare și kerneluri**

PixInsight. *Interpolation Algorithms.* Link

**Înțelegere și concepte de interpolare în procesarea imaginilor**

*Computerphile – What is Image Interpolation?* Video

*Computerphile – How Does Interpolation Work?* Video

**Ideea de implementare a codului și conceptul algoritmului Lanczos**

*M – Lanczos Algorithm Explained.* Video

**Teorie Lanczos**

*Edgar Programmator – Lanczos Interpolation Theory.* Video

**Bicubic Interpolation**

Wikipedia – *Bicubic Interpolation and Bicubic Convolution Algorithm.* Link

**Imbunătățiri**

OpenCV Docs – How to use the OpenCV parallel_for_ to parallelize your code. Link