Algorytmy macierzowe
Laboratorium 3

Marcel Duda, Jan Gawroński

12.01.2026

1 Kod

```
TreeNode* createTree(const Matrix &A, int rank, double epsilon) {
  auto [U, D, V] = svd_decomposition(A, rank);

  if (static_cast<int>(D.size()) < rank || D[rank - 1] < epsilon) {
    if (allclose_zero(A, 1e-10)) {
      TreeNode* node = new TreeNode();
      node->singularValues = zeroMatrix(1, rank)[0];
      node->U = zeroMatrix(rows(A), rank);
      node->V = zeroMatrix(rank, cols(A));
      return node;
    }
    TreeNode* node = new TreeNode();
    node->singularValues = D;
    node->U = subMatrix(U, 0, 0, rows(U), static_cast<int>(D.size()));
    node->V = subMatrix(V, 0, 0, static_cast<int>(D.size()), cols(V));
    return node;
  }
  TreeNode* node = new TreeNode();
  int rmid = rows(A) / 2;
  int cmid = cols(A) / 2;
  node->topLeft = createTree(subMatrix(A, 0, 0, rmid, cmid), rank, epsilon);
  node->topRight = createTree(subMatrix(A, 0, cmid, rmid, cols(A) - cmid), rank, epsilon);
  node->bottomLeft = createTree(subMatrix(A, rmid, 0, rows(A) - rmid, cmid), rank, epsilon);
  node->bottomRight = createTree(subMatrix(A, rmid, cmid, rows(A) - rmid, cols(A) - cmid), rank, epsilon);
  return node;
}
```

```
27
28 Matrix reconstructFromTree(TreeNode* node) {
29   if (node->topLeft == nullptr && node->topRight == nullptr &&
30     node->bottomLeft == nullptr && node->bottomRight == nullptr) {
31     int r = static_cast<int>(node->singularValues.size());
32     Matrix S = zeroMatrix(r, r);
33     for (int i = 0; i < r; ++i) S[i][i] = node->singularValues[i];
34     return node->U * S * node->V;
35   }
36   Matrix A11 = reconstructFromTree(node->topLeft);
37   Matrix A12 = reconstructFromTree(node->topRight);
38   Matrix A21 = reconstructFromTree(node->bottomLeft);
39   Matrix A22 = reconstructFromTree(node->bottomRight);
40   return combine(A11, A12, A21, A22);
41 }
42
43
44 std::pair<Vector, double> power_iteration(const Matrix &A, int
       num_simulations) {
45   size_t n = A.size();
46   std::mt19937_64 gen(std::random_device{}());
47   std::uniform_real_distribution<double> dist(0.0, 1.0);
48   Vector b_k(n);
49   for (size_t i = 0; i < n; ++i) b_k[i] = dist(gen);
50
51   for (int it = 0; it < num_simulations; ++it) {
52     Vector b_k1 = mat_vec_mul(A, b_k);
53     double norm = vec_norm(b_k1);
54     if (norm == 0.0) break;
55     for (size_t i = 0; i < n; ++i) b_k[i] = b_k1[i] / norm;
56   }
57   Vector Ab = mat_vec_mul(A, b_k);
58   double denom = vec_dot(b_k, b_k);
59   double eigenvalue = denom == 0.0 ? 0.0 : vec_dot(b_k, Ab) /
         denom;
60   return {b_k, eigenvalue};
61 }
62
63
64 std::tuple<Matrix, Vector, Matrix> svd_decomposition(const Matrix
       &A, int r, double epsilon) {
65   size_t m = A.size(), n = A.empty() ? 0 : A[0].size();
66   Matrix U(m, Vector(r, 0.0));
67   Matrix V(r, Vector(n, 0.0));
68   Vector S;
69   Matrix B = transpose_mul(A);
70   int num_valid = 0;
71
72   for (int i = 0; i < r; ++i) {
```

```
    auto [v, sigma_squared] = power_iteration(B);
    if (sigma_squared < epsilon * epsilon) break;
    double sigma = std::sqrt(sigma_squared);
    S.push_back(sigma);
    if ((size_t)i < V.size()) V[i] = v;
    Vector u = mat_vec_mul(A, v);
    if (sigma != 0.0) for (double &x : u) x /= sigma;
    for (size_t r = 0; r < m; ++r) U[r][num_valid] = u[r];
    for (size_t p = 0; p < n; ++p)
        for (size_t q = 0; q < n; ++q)
            B[p][q] -= sigma_squared * v[p] * v[q];
    ++num_valid;
    if (allclose_zero(B, epsilon)) break;
  }

  Matrix U_trim(m, Vector(num_valid));
  Matrix V_trim(num_valid, Vector(n));
  for (size_t r = 0; r < m; ++r)
    for (int c = 0; c < num_valid; ++c)
      U_trim[r][c] = U[r][c];
  for (int r = 0; r < num_valid; ++r)
    for (size_t c = 0; c < n; ++c)
      V_trim[r][c] = V[r][c];

  return {U_trim, S, V_trim};
}
```

2 Wartości osobliwe

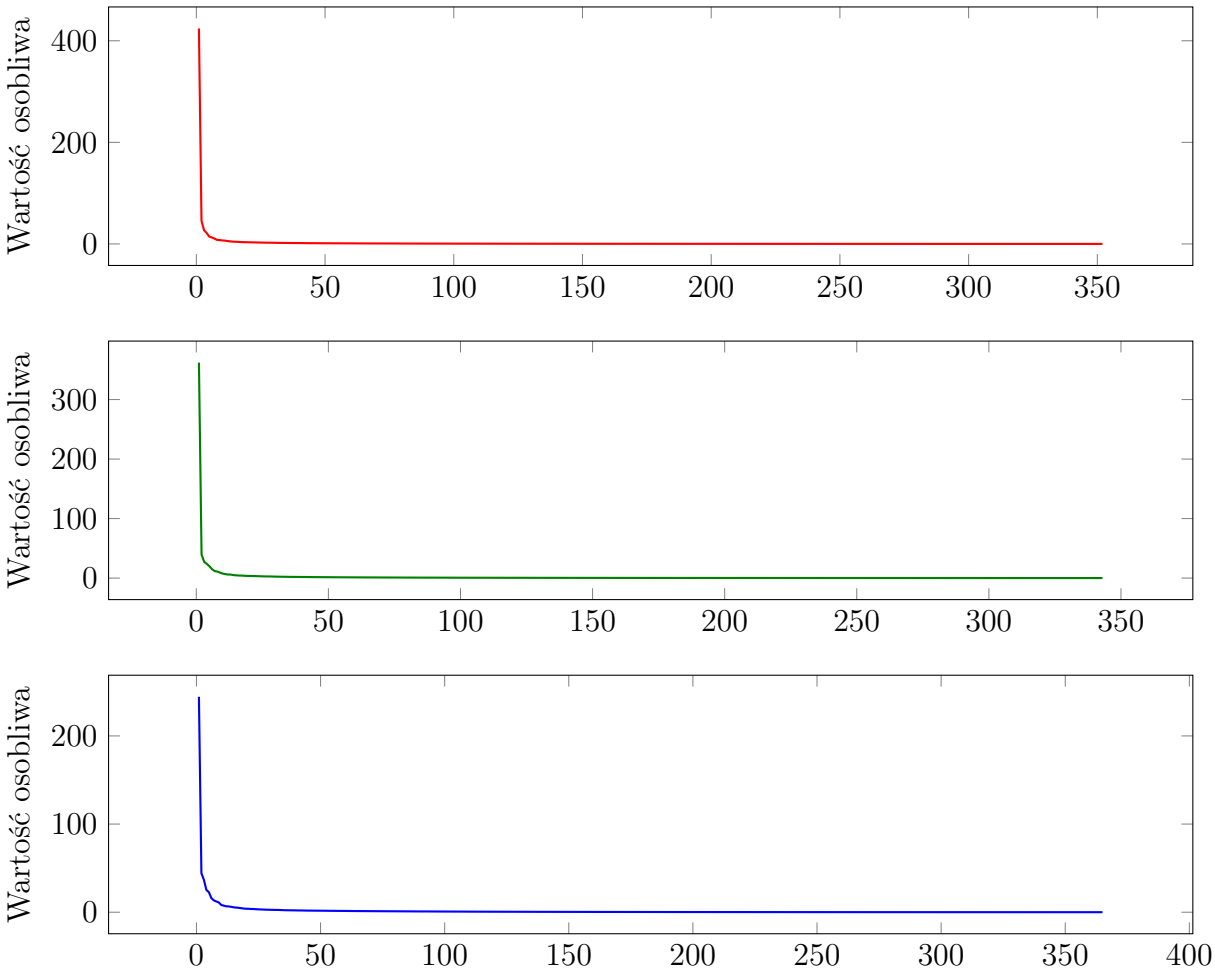

Figure 2.0.1: Wartości osobliwe dla kanałów R, G, B.

3 Obrazy

doge.png

dogeR.png

dogeG.png

dogeB.png

Figure 3.0.1: Doge oryginalny oraz obrazy kanałów R/G/B.

4 Kompresje

r = 2, e = 0.000000

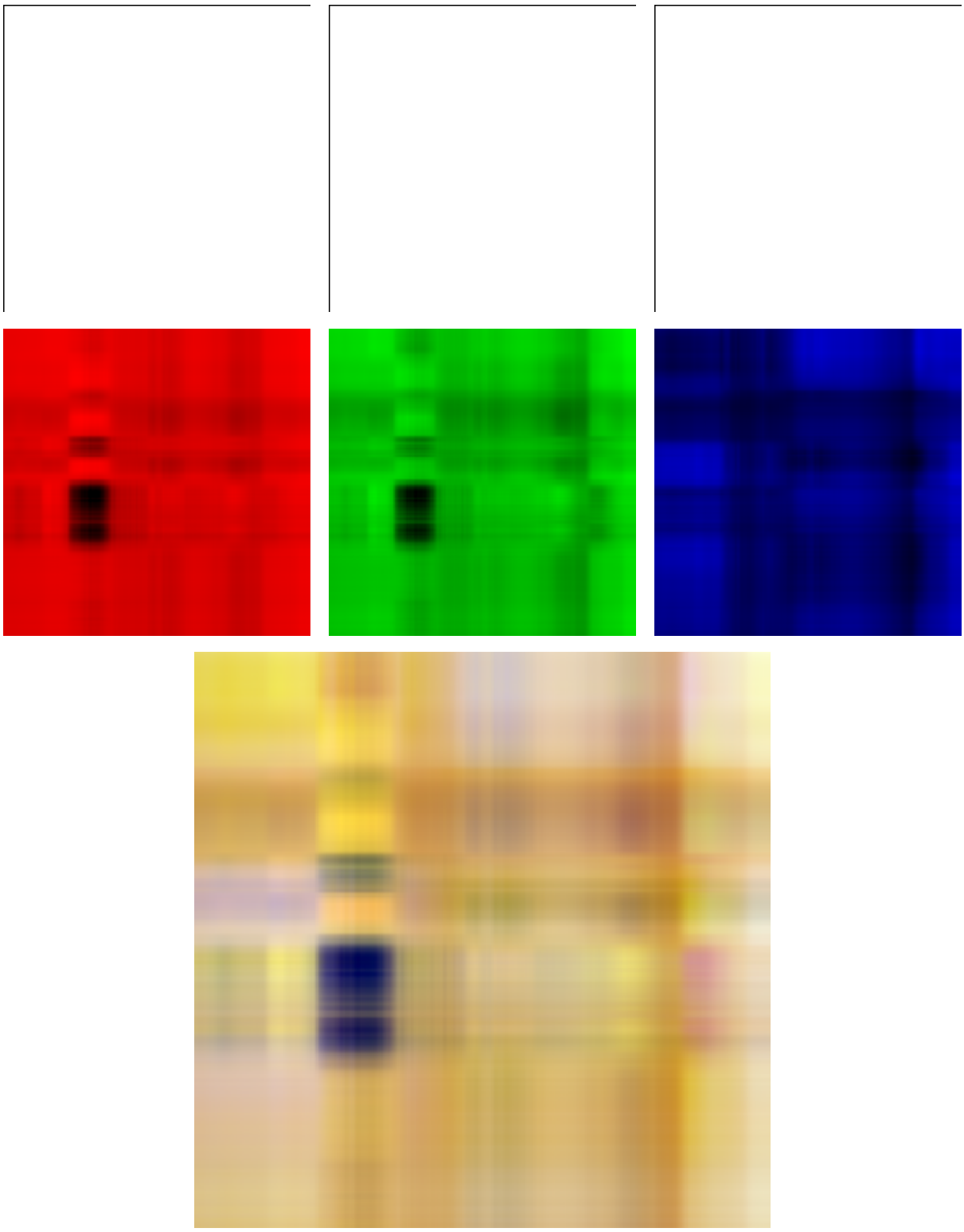

Figure 4.0.1: Kompresja dla $r = 2$, $e = 0.000000$. Góra: macierze skompresowane (R,G,B). Środek: odtworzone kanały R,G,B. Dół: obraz RGB złożony.

r = 2, e = 0.500000

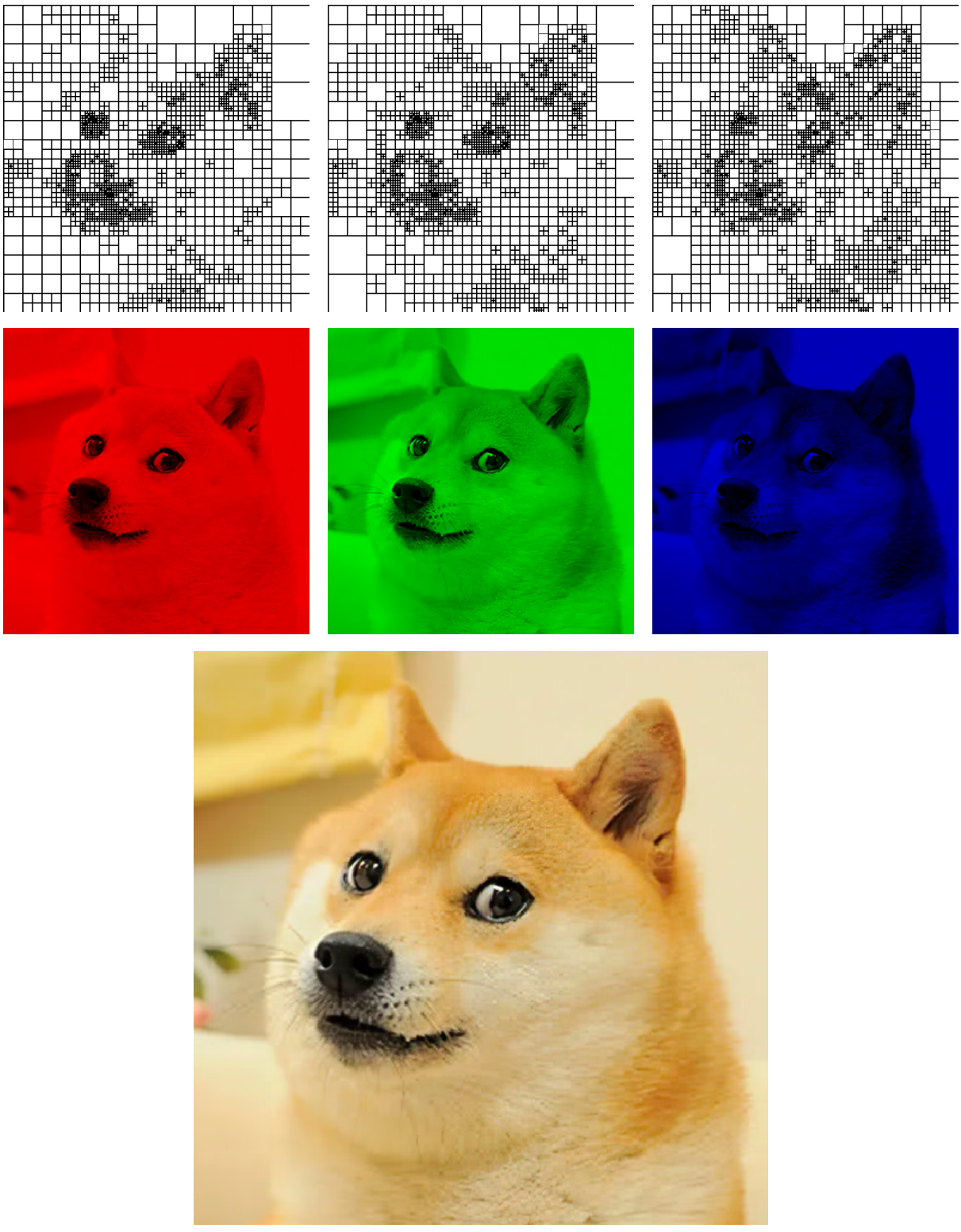

Figure 4.0.2: Kompresja dla $r = 2$, $e = 0.500000$. Góra: macierze skompresowane (R,G,B). Środek: odtworzone kanały R,G,B. Dół: obraz RGB złożony.

r = 2, e = 1.000000

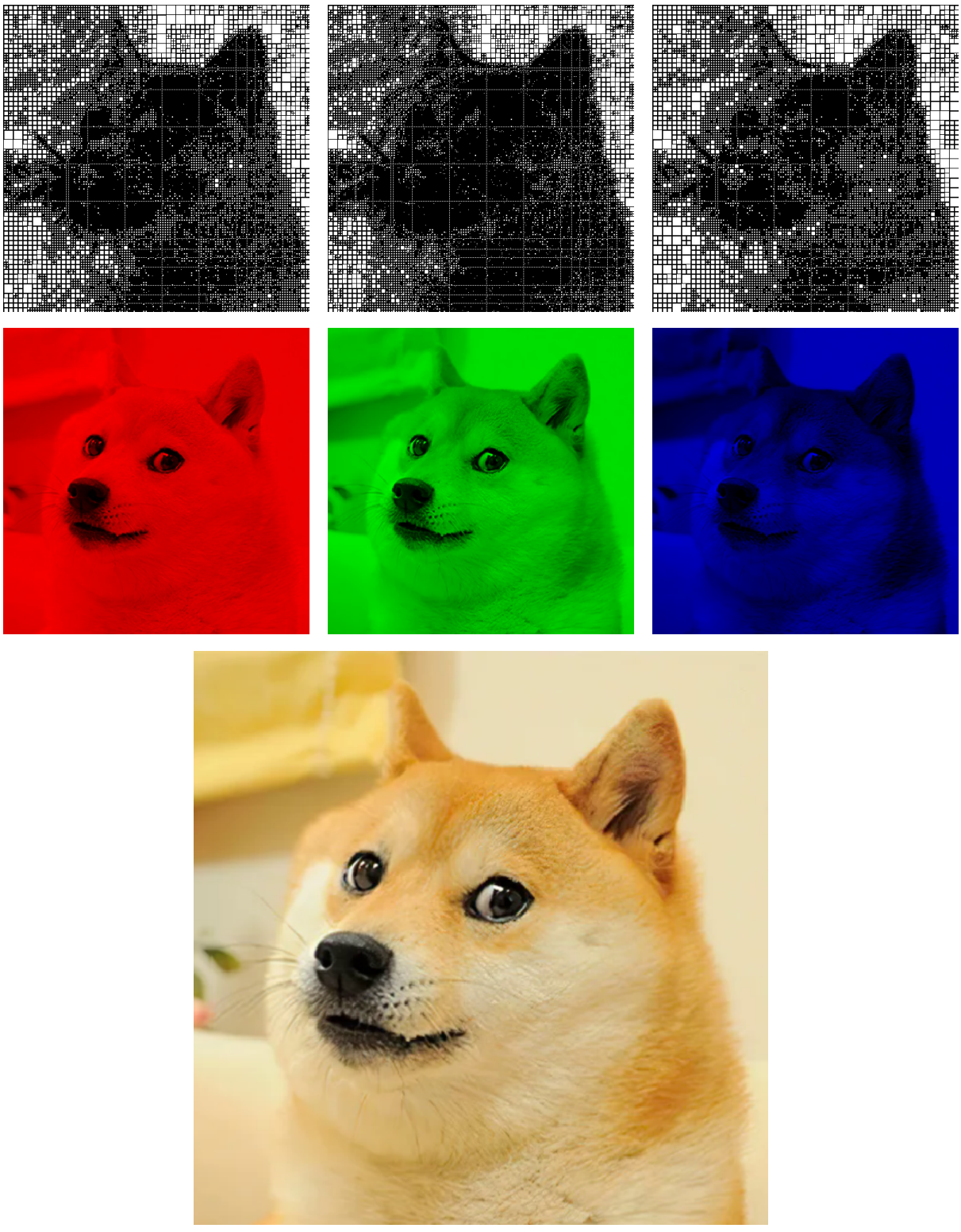

Figure 4.0.3: Kompresja dla $r = 2$, $e = 1.000000$. Góra: macierze skompresowane (R,G,B). Środek: odtworzone kanały R,G,B. Dół: obraz RGB złożony.

r = 4, e = 0.000000

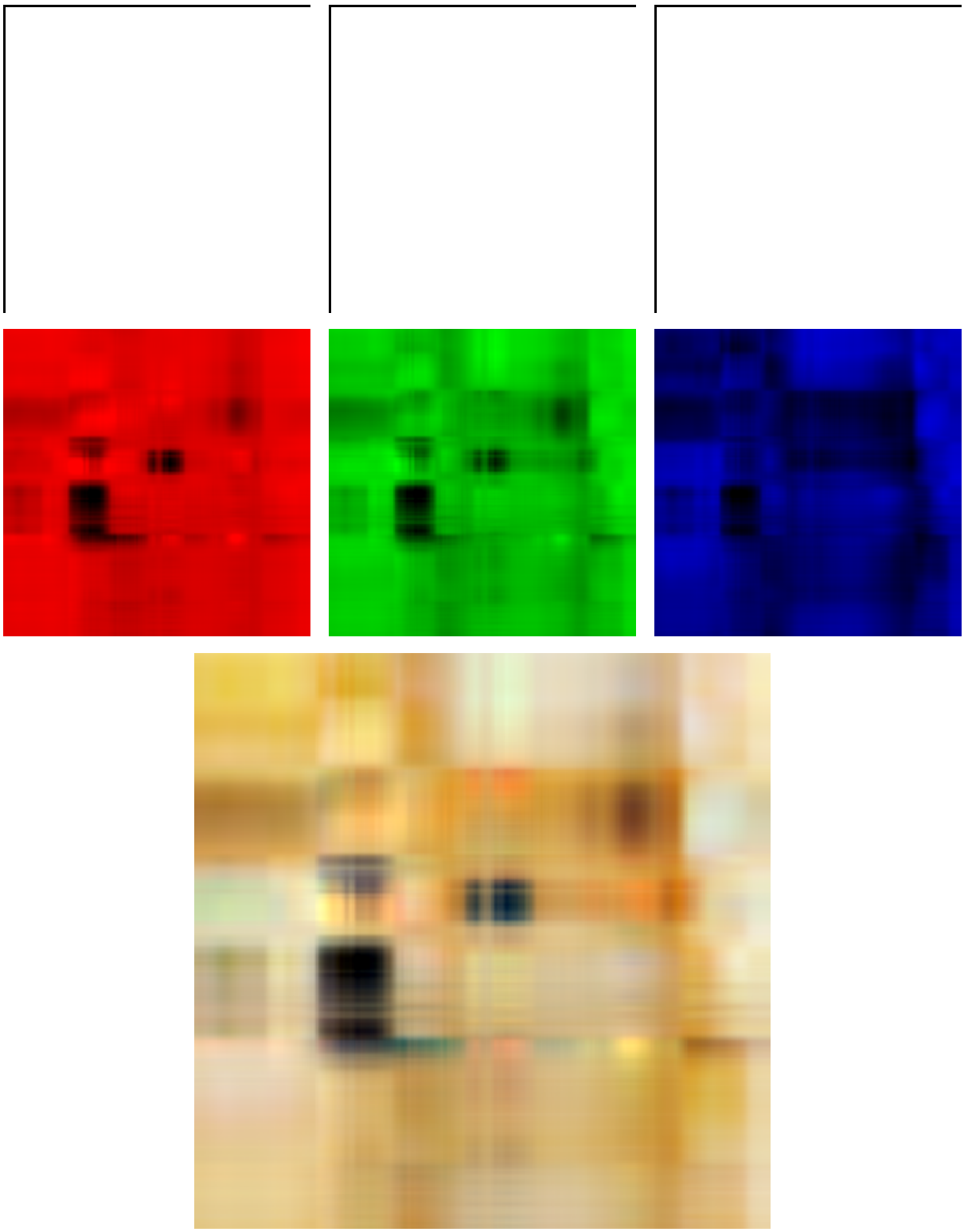

Figure 4.0.4: Kompresja dla $r = 4$, $e = 0.000000$. Góra: macierze skompresowane (R,G,B). Środek: odtworzone kanały R,G,B. Dół: obraz RGB złożony.

r = 4, e = 0.500000

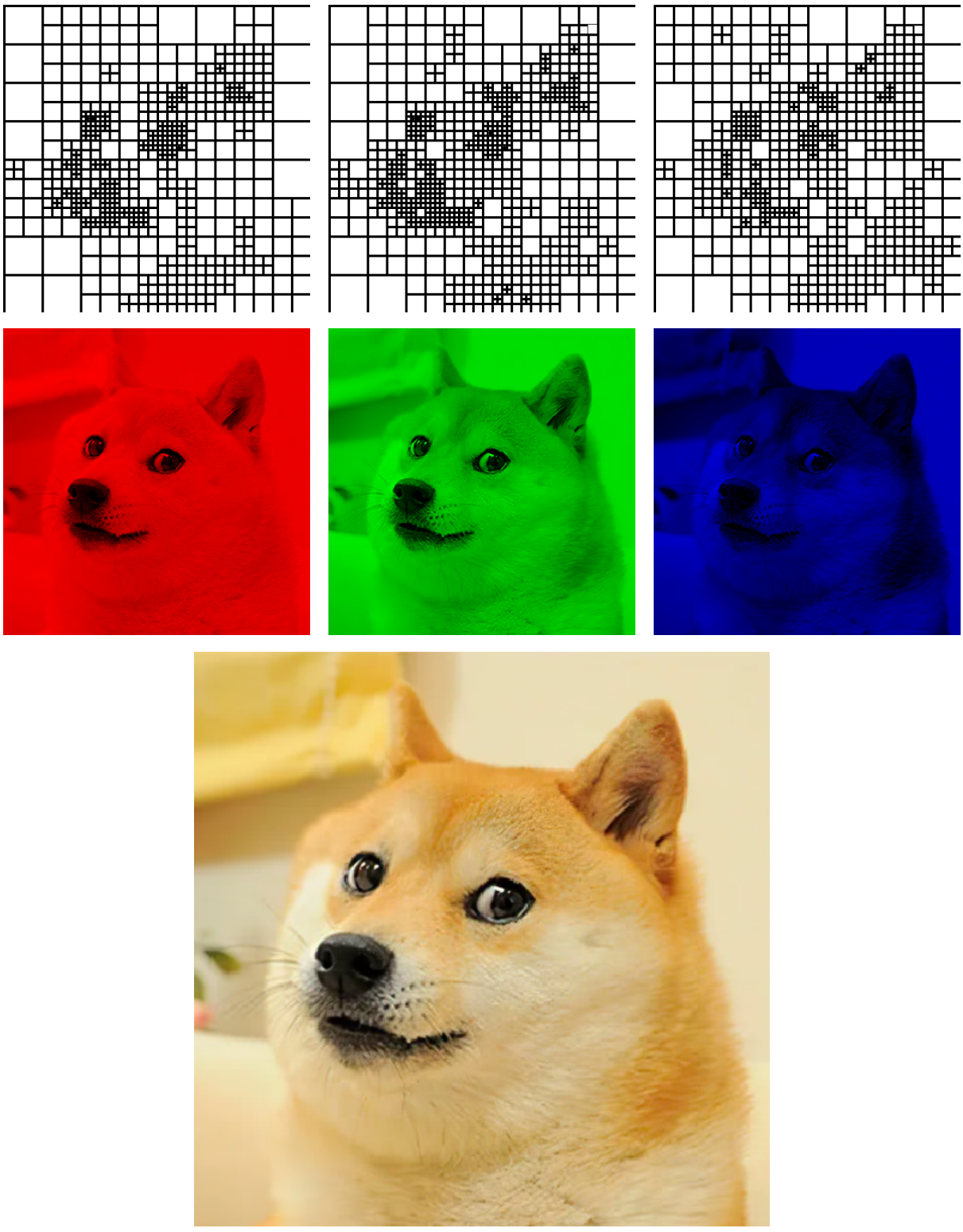

Figure 4.0.5: Kompresja dla $r = 4$, $e = 0.500000$. Góra: macierze skompresowane (R,G,B). Środek: odtworzone kanały R,G,B. Dół: obraz RGB złożony.

r = 4, e = 1.000000

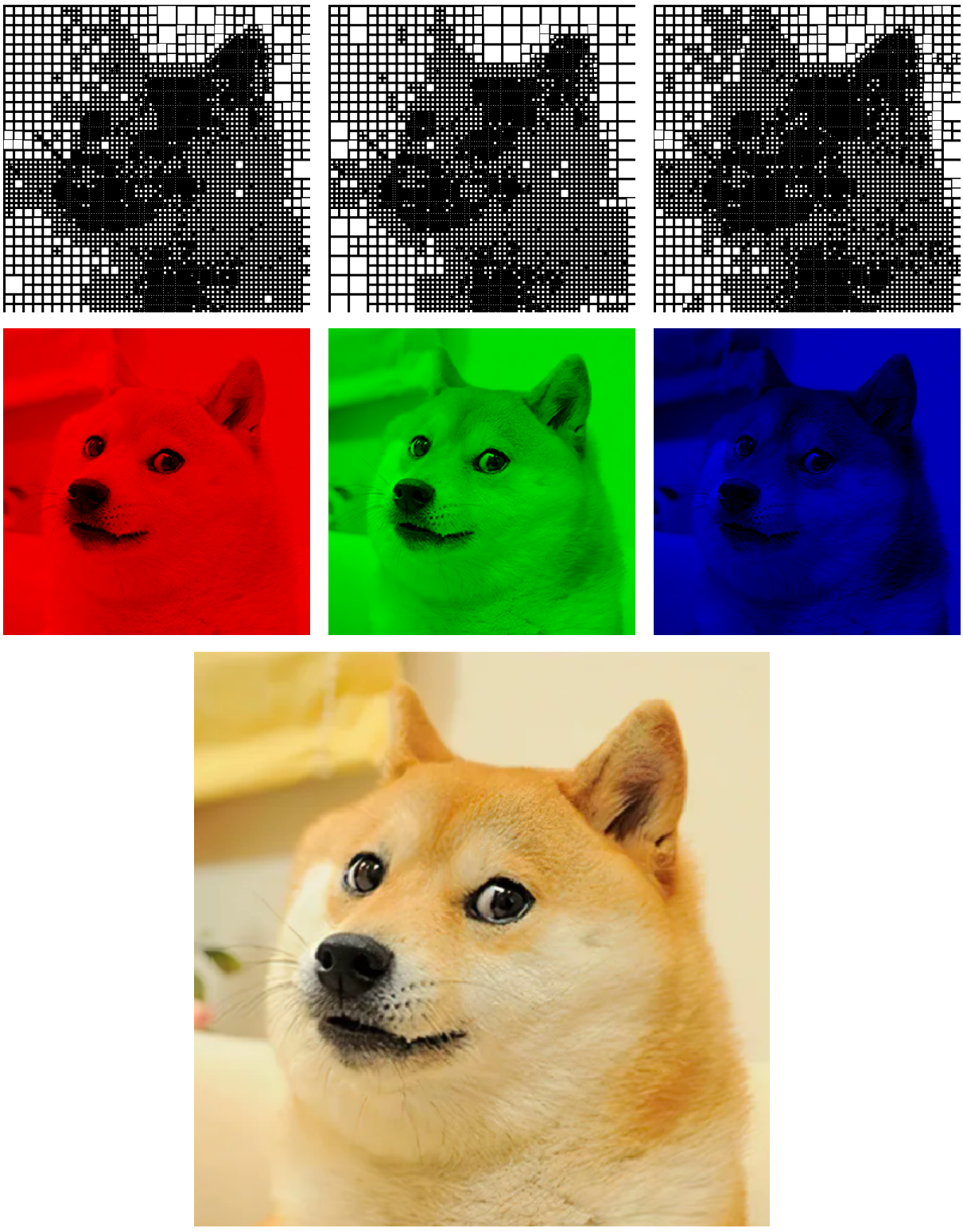

Figure 4.0.6: Kompresja dla $r = 4$, $e = 1.000000$. Góra: macierze skompresowane (R,G,B). Środek: odtworzone kanały R,G,B. Dół: obraz RGB złożony.