Metody Numeryczne N02

Jakub Kurek

1. Wstęp

Celem zadania jest sprawdzenie różnych metod rozwiązania układu równań $Au = b$

$$A^{N\times N} = \begin{pmatrix} \frac{-2}{h^2} & \frac{1}{h^2} & & & \frac{1}{h^2} \\ \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & & \\ & \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & \\ & & & \ddots & \\ \frac{1}{h^2} & & & \frac{1}{h^2} & \frac{-2}{h^2} \end{pmatrix} \qquad h = \frac{2}{N-1} \qquad b_n = \cos\left(\frac{4\pi(n-1)}{N}\right)$$

Program liczący rozwiązania został napisany w C++23 przy użyciu biblioteki ***Eigen*** w wersji 5.0.1. Wykresy zostały stworzone w języku Python przy użyciu biblioteki matplotlib. Cały kod wykorzystywany do obliczeń oraz generowania wykresów znajduje się w repozytorium na ***GitHub***.

1.1. Konfiguracja sprzętowa

- Procesor: Intel i7-8650U (8) @ 4.200GHz
- Pamięć RAM: 32GB DDR4 2133MHz
- System operacyjny: Arch Linux 6.17.5-arch1-1

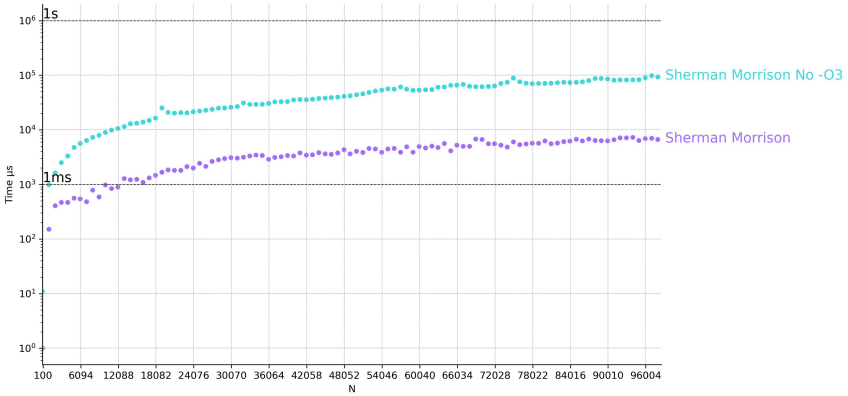
1.2. Metodyka testowania

Wszystkie testy były wykonywane na odciążonym procesorze bezpośrednio z powłoki. Dla obliczeń trwających poniżej 20ms testy były wykonywane 10-krotnie a czas był uśredniany. Pomiary czasu były wykonywane w mikrosekundach w celu lepszego zobrazowania czasu obliczeń dla małych macierzy oraz optymalnych metod. Po wprowadzeniu optymalizacje wszystkie dalsze testy były kompilowane przy użyciu kompilatora clang++ w wersji 21.1.4 z flagami `-Wall -Wextra -std=c++23 -pedantic -I ./../eigen-5.0.1/ -O3 -march=native -DNDEBUG`.

2. Wstępne optymalizacje

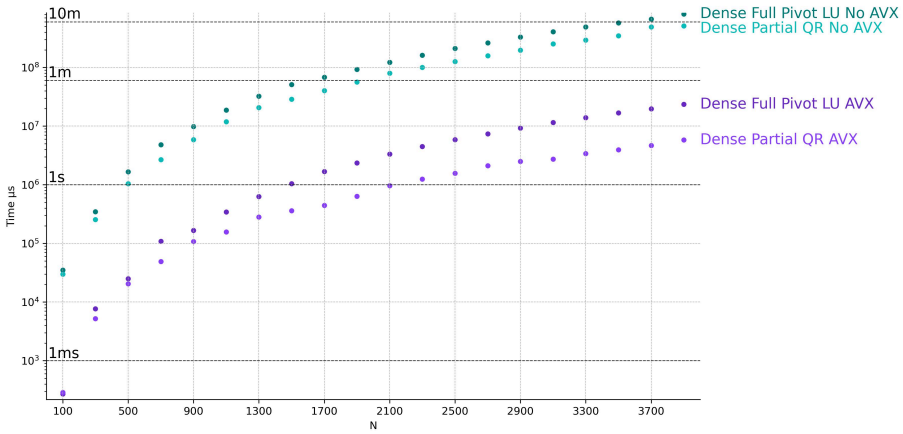
Najważniejszą optymalizacją poprawiające wyniki obliczeń było zastosowanie odpowiednich flag kompilatora:

- `-O3` - poziom optymalizacji kodu przez kompilator. Wykres 1 pokazuje róznice czasu wykonania zwiększoną około 10-krotnie. W przypadku funkcji bibliotecznych ta różnica zmniejsza się do około 2-krotności.



Wykres 1: Porównanie algorytmu Thomasa z wzorem Shermana-Morrisona z optymalizacją -O3 i bez

- `-march=native` - możliwośc wykonywania instrukcji wektoryzujących AVX. Wykres 2 pokazuje róznice czasu obliczeń wybranych algorytmów z bilbioteki Eigen po włączeniu nstrukcji AVX.



Wykres 2: Porównanie algorytmów z biblioteki Eigen z użyciem instrukcji AVX i bez

- `-DNDEBUG` - wyłączenie assercji. Eigen dla macierzy o dynamicznych rozmiarach sprawdza poprawność ich rozmiaru przy użyciu assercji. Nie ma wielkiego wpływu na pojedyncze działanie programu dla dużych macierzy, lecz może wpływać kiedy wykonujemy wiele działań na wektorach lub macierzach. Zalecane przez dokumentacje.

3. Rozwiązania metodami gęstymi

Biblioteka Eigen oferuje moduł służący do rozwiązywania układów równań przy użyciu macierzy gęstych. Wybrane zostały metody:

- Full Pivot LU
- Partial Pivot LU
- Householder Full Pivot QR
- Householder Partial Pivot QR

Wszystkie metody uzywały wspólnej funkcji `gen_desnse_A()` (Kod 1) do generowania gęstej macierzy oraz funkcji `gen_b_vector()` (Kod 2) do tworzenia wektora rowiązania. Funkcja `gen_dense_B` była wykorzystywana do sprawdzania działania własnej implementacji alorytmu Thomasa, co wymusiło rozbicie tworzenia macierzy gęstej na dwa etapy.

```cpp
Eigen::MatrixXd gen_dense_B(long N) {
  // matrix must be greater than 1
  // assert(N != 0);
  float64_t h = 2 / ((float64_t)N - 1.0);
  Eigen::MatrixXd mat = Eigen::MatrixXd::Zero(N, N);

  for (long x = 1; x < N - 1; x++) {
    mat(x, x - 1) = 1.0 / (h * h);
    mat(x, x + 1) = 1.0 / (h * h);
    mat(x, x) = -2.0 / (h * h);
  }
  mat(0, 0) = -3.0 / (h * h);
  mat(N - 1, N - 1) = -3.0 / (h * h);

  // Diagonal fix
  mat(0, 1) = 1.0 / (h * h);
  mat(N - 1, N - 2) = 1.0 / (h * h);

  return mat;
}

// A = B + uvT
Eigen::MatrixXd gen_dense_A(long N) {
  auto mat = gen_dense_B(N);
  float64_t h = 2 / ((float64_t)N - 1.0);

  // Distortion from 3 diagonal
  mat(0, N - 1) = 1.0 / (h * h);
  mat(N - 1, 0) = 1.0 / (h * h);

  mat(0, 0) += 1.0 / (h * h);
  mat(N - 1, N - 1) += 1.0 / (h * h);
  return mat;
}
```

Kod 1: Generowanie gęstej macierzy pełnej oraz samej wersji diagonalnej

```cpp
Eigen::VectorXd gen_b_vector(long N) {
  Eigen::VectorXd b = Eigen::VectorXd(N);
  b.setZero();
  for (long n = 1; n <= N; n++) {
    float64_t val = std::cos((4.0 * std::numbers::pi * (n - 1)) / (double)N);
    b(n - 1) = val;
  }

  return b;
}
```

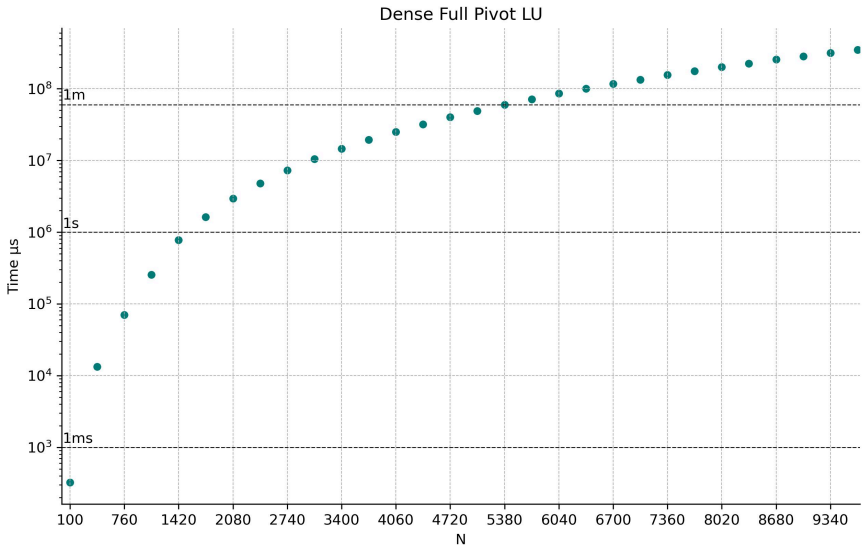
Kod 2: Generowanie wektora rozwiązania

3.1. Full Pivot LU

Pierwszą z omawianych metod jest rozwiązanie układu równań przy użyciu dekompozycji LU z pełnym pivotingiem. Analizując wybrane dane z Tabela 1 metoda rośnie w czasie $O(N^3)$ co jest zgodne z teoretyczna złozonością algorytmu. Jest to bazowy wynik, do którego kolejne metody będą się odnosić.

```
Eigen::VectorXd solve_d_mat_fullpiv_lu(long N) {
  Eigen::MatrixXd d_mat = gen_dense_A(N);
  Eigen::VectorXd b = gen_b_vector(N);

  Eigen::VectorXd u = d_mat.fullPivLu().solve(b);
  return u;
}
```

Kod 3: Rozwiązanie równania przy użyciu rozkładu LU z pełnym pivotem dla podanego N



Wykres 3: Czas rozwiązania $Au = b$ dla różnych N przy użyciu fullPivLU

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$324\mu s$	$254ms$	$2947ms$	$10s$	$25s$	$49s$	$86s$	$134s$	$200s$	$283s$	$349s$

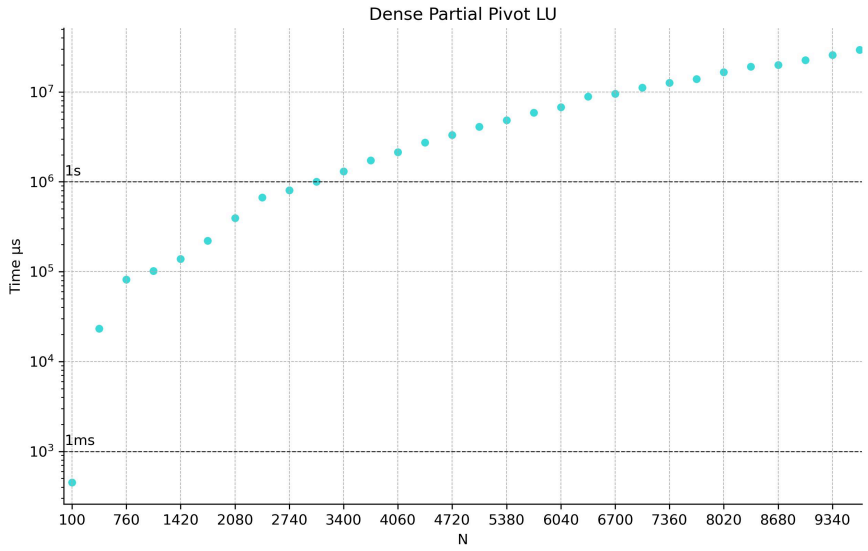
Tabela 1: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu fullPivLU

3.2. Partial Pivot LU

Kolejną z omawianych metod jest rozwiązanie układu równań przy użyciu dekompozycji LU z częściowym pivotingiem. Kosztem precyzji numerycznej zyskujemy krótszy czas wykonania. Analizując wybrane dane z Tabela 2 metoda rośnie w czasie $O(N^3)$, lecz w porównaniu z czasami dla pełnego pivotingu (Tabela 1) czas rozwiązywania układu równań jest ponad 12-krotnie mniejszy.

```
Eigen::VectorXd solve_d_mat_partialpiv_lu(long N) {
  Eigen::MatrixXd d_mat = gen_dense_A(N);
  Eigen::VectorXd b = gen_b_vector(N);

  Eigen::VectorXd u = d_mat.partialPivLu().solve(b);
  return u;
}
```

Kod 4: Rozwiązanie równania przy użyciu rozkładu LU z częściowym pivotingiem



Wykres 4: Czas rozwiązania $Au = b$ dla różnych N z częściowym pivotem dla podanego N przy użyciu partialPivLU

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$451\mu s$	$102ms$	$395ms$	$999ms$	$2138ms$	$4090ms$	$6753ms$	$11s$	$16s$	$22s$	$29s$

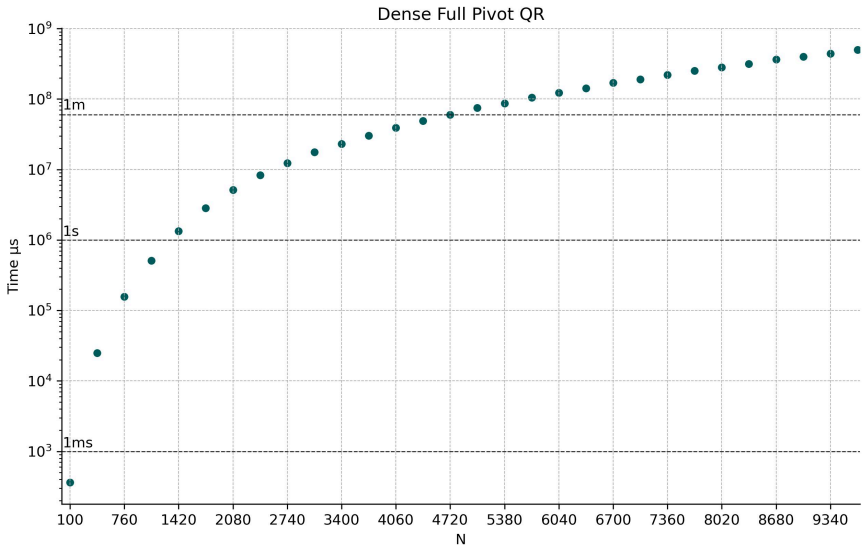
Tabela 2: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu partialPivLU

3.3. Householder Full Pivot QR

Następną omawianą metodą jest dekompozycja QR z pełnym pivotingiem. Analizując wybrane dane z Tabela 3 metoda rośnie w czasie $O(N^3)$ i potwierdza to teoretyczną złożoność algorytmu. W porównaniu z czasami rozkładu LU z pełnym pivotingiem (Tabela 1) jest około 1.4 razy wolniejsza. W naszym przypadku rozwiązywania pojedynczego równania, rozkład QR nie oferuje wystarczających korzyści w odniesieniu do poniesionych kosztów.

```
Eigen::VectorXd solve_d_mat_fullpiv_qr(long N) {
  Eigen::MatrixXd d_mat = gen_dense_A(N);
  Eigen::VectorXd b = gen_b_vector(N);

  Eigen::VectorXd u = d_mat.fullPivHouseholderQr().solve(b);
  return u;
}
```

Kod 5: Rozwiązanie równania przy użyciu rozkładu QR z pełnym pivotem dla podanego N



Wykres 5: Czas rozwiązania $Au = b$ dla różnych N przy użyciu fullPivHouseholderQr

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$363\mu s$	$512ms$	$5169ms$	$17s$	$39s$	$74s$	$122s$	$191s$	$282s$	$399s$	$499s$

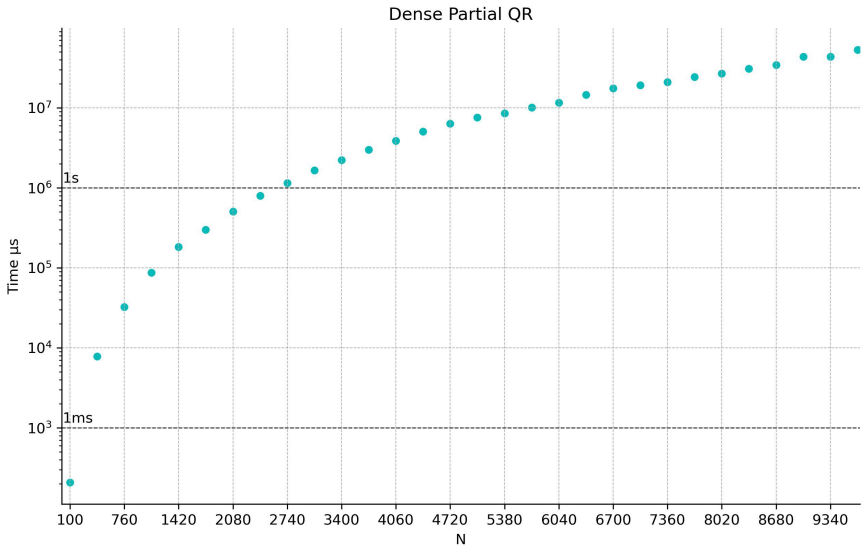
Tabela 3: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu fullPivHouseholderQr

3.4. Householder Partial Pivot QR

Ostatnia omawianą metoda dla macierzy gęstych jest dekompozycja QR z częściowym pivotingiem. Analizując czasy z Tabela 4 metoda rośnie w czasie $O(N^3)$. Prównując ją z jej odpowiednikiem, czyli rozkładem LU z częściowym pivotingiem (Tabela 2) jest ona około 1.8 razy wolniejsza. Jak w przypadku dekompozycji z pełnym pivotingiem nie oferuje ona wystarczających zysków w stosunku odpowiadającej jej metodzie z rozkładem LU.

```
Eigen::VectorXd solve_d_mat_partialpiv_qr(long N) {
  Eigen::MatrixXd d_mat = gen_dense_A(N);
  Eigen::VectorXd b = gen_b_vector(N);

  Eigen::VectorXd u = d_mat.householderQr().solve(b);
  return u;
}
```

Kod 6: Rozwiązanie równania przy użyciu rozkładu QR z częściowym pivotem dla podanego N



Wykres 6: Czas rozwiązania $Au = b$ dla różnych N przy użyciu householderQr

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$208\mu s$	$87ms$	$506ms$	$1663ms$	$3869ms$	$7588ms$	$11s$	$19s$	$26s$	$43s$	$53s$

Tabela 4: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu partialPivQR

4. Rozwiązania metodami dla macierzy rzadkich

Biblioteka Eigen oferuje również moduł służący do rozwiązywania układów równań przy użyciu macierzy rzadkich. Nasza macierz jest trójdagonalna z 2 elementami poza nimi. Dla dużych N jest ona w większości wypełniona zerami, co może zostać wykorzystane poprzez przedstawienie jej w reprezentacji rzadkiej. Wybrane zostały metody:

- Sparse LU
- Sparse QR

Obie metody używały wspólnej funkcji `gen_sparse_A()` (Kod 7) do generowania rzadkiej macierzy. Metody uzywają gęstego wektora generowanego przez Kod 2, gdyż jest on całkowicie zapełniony.

```cpp
Eigen::SparseMatrix<double> gen_sparse_A(long N) {
 float64_t h = 2 / ((float64_t)N - 1.0);

 Eigen::SparseMatrix<double> sp_mat = Eigen::SparseMatrix<double>(N, N);
 std::vector<Eigen::Triplet<double>> tripletList =
     std::vector<Eigen::Triplet<double>>(N);

 for (long x = 1; x < N - 1; x++) {
   tripletList.push_back(Eigen::Triplet<double>(x, x, -2.0 / (h * h)));
   tripletList.push_back(Eigen::Triplet<double>(x, x + 1, 1.0 / (h * h)));
   tripletList.push_back(Eigen::Triplet<double>(x, x - 1, 1.0 / (h * h)));
 }

 // Diagonal fix
 tripletList.push_back(Eigen::Triplet<double>(0, 1, 1.0 / (h * h)));
 tripletList.push_back(Eigen::Triplet<double>(N - 1, N - 2, 1.0 / (h * h)));
 tripletList.push_back(Eigen::Triplet<double>(0, 0, -2.0 / (h * h)));
 tripletList.push_back(Eigen::Triplet<double>(N - 1, N - 1, -2.0 / (h * h)));

 // Corners
 tripletList.push_back(Eigen::Triplet<double>(N - 1, 0, 1.0 / (h * h)));
 tripletList.push_back(Eigen::Triplet<double>(0, N - 1, 1.0 / (h * h)));

 sp_mat.setFromTriplets(tripletList.begin(), tripletList.end());
 sp_mat.makeCompressed();

 return sp_mat;
}
```

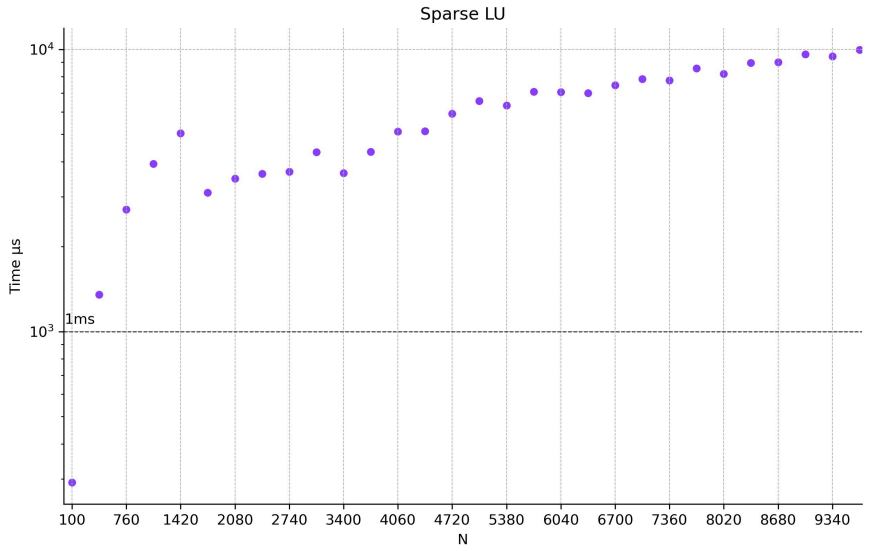
Kod 7: Generowanie rzadkiej macierzy

4.1. Sparse LU

Pierwszą z opisywanych metod, które próbują wykorzystać wcześniej znaną strukture macierzy jest rozkład LU działający tylko na nie zerowych elementach macierzy. Dla $N < 10000$ rozkład skaluje się w przybliżeniu liniowo $O(N)$ po analizie czasów z Tabela 5. Dokładniejsza analiza dla $N > 10000$ zostanie przeprowadzona wspólnie z resztą względnie efektywnych metod. Dla $N < 10000$ mimo uśredniania czasów dalej duży wpływ na efektywnośc metody posiada system operacyjny i przełączanie zadań. Wstępnie porównując czasy dla obecnie najbardziej efektywnej metody, czyli rozkładu LU z częściowym pivotingiem dla macierzy gęstej (Tabela 2), wybranie metody rzadkiej powoduje zwiększenie wydajności dla macierzy $N = 9010$ o około 2200 razy.

```cpp
Eigen::VectorXd solve_sp_mat_lu(long N) {
  auto sp_mat = gen_sparse_A(N);
  auto b = gen_b_vector(N);

  Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>>
      sparse_lu_solver;
  sparse_lu_solver.analyzePattern(sp_mat);
  sparse_lu_solver.factorize(sp_mat);

  auto u = sparse_lu_solver.solve(b);
  return u;
}
```

Kod 8: Rozwiązanie równania przy użyciu rozkładu LU wykorzystującego rzadką postac macierzy



Wykres 7: Czas rozwiązania $Au = b$ dla różnych N przy użyciu sparse LU

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$292\mu s$	$3925\mu s$	$3484\mu s$	$4324\mu s$	$5113\mu s$	$6554\mu s$	$7048\mu s$	$7856\mu s$	$8193\mu s$	$9594\mu s$	$9959\mu s$

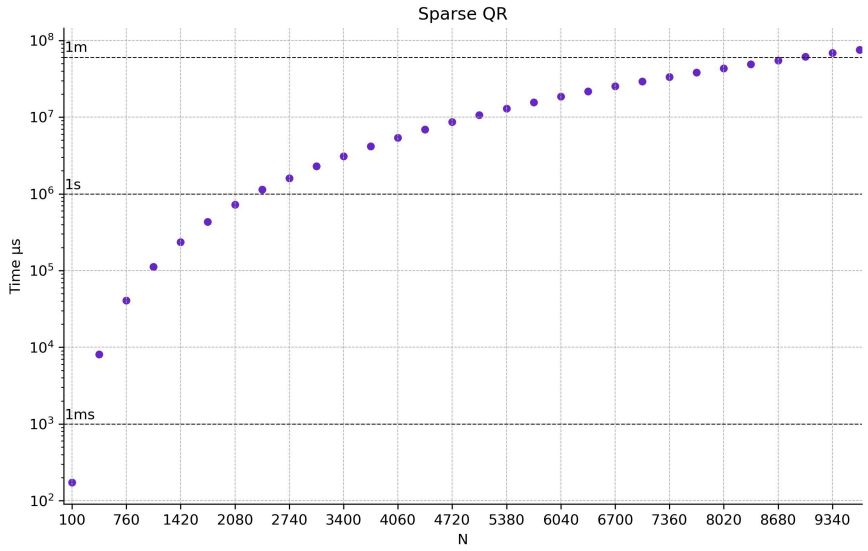
Tabela 5: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu sparse LU

4.2. Sparse QR

Drugą z oferowanych przez bibliotekę Eigen metod dekompozycji macierzy rzadkich jest zmodyfikowany rozkład QR. Nasza macierz mimo rzadkiej reprezentacji źle wpływa na rzadki rozkład QR i powoduje, że czasy rozwiązań z Tabela 6 są gorsze niż dla gęstych rozkładów QR i LU z częściowym pivotingiem. W porównaniu z gęstym QR z częściowym pivotingiem (Tabela 4) jest on około 1.5 razy wolniejsza. Jest to dobry przykład, że mimo rzadkiej struktury nie zawsze rzadki algorytm będzie lepszym rozwiązaniem dla czasu rozwiązania.

```
Eigen::VectorXd solve_sp_mat_lu(long N) {
  auto sp_mat = gen_sparse_A(N);
  auto b = gen_b_vector(N);

  Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>>
      sparse_lu_solver;
  sparse_lu_solver.analyzePattern(sp_mat);
  sparse_lu_solver.factorize(sp_mat);

  auto u = sparse_lu_solver.solve(b);
  return u;
}
```

Kod 9: Rozwiązanie równania przy użyciu rozkładu QR wykorzystującego rzadką postac macierzy



Wykres 8: Czas rozwiązania $Au = b$ dla różnych N przy użyciu sparse QR

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$173\mu s$	$112ms$	$722ms$	$2290ms$	$5425ms$	$10s$	$18s$	$29s$	$43s$	$61s$	$75s$

Tabela 6: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu sparse QR

5. Rozwiązanie wykorzystujące strukture macierzy

5.1. Obserwacje

Analizując postać macierzy A można zauważyć, że jest ona trójdiagonalna z dodatkowymi elementami w pozycjach $(1, N)$ oraz $(N, 1)$. Korzystając z obserwacji można zapisać macierz A jako $A = B + uv^\mathsf{T}$, wówczas:

$$B^{N\times N} = \begin{pmatrix} 2 * \frac{-2}{h^2} & \frac{1}{h^2} & & & & \\ \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & & & \\ & \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & & \\ & & & \ddots & & \\ & & & \frac{1}{h^2} & \frac{-2}{h^2} - \left(\frac{1}{2h^2}\right) \end{pmatrix} \quad u = \begin{pmatrix} \frac{2}{h^2} \\ 0 \\ \vdots \\ 0 \\ \frac{1}{h^2} \end{pmatrix} \quad v = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \frac{1}{2} \end{pmatrix} uv^\mathsf{T} = \begin{pmatrix} \frac{2}{h^2} & 0 & \dots & 0 & \frac{1}{h^2} \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 0 & 0 \\ \frac{1}{h^2} & 0 & \dots & 0 & \frac{1}{2h^2} \end{pmatrix}$$

Pozwoli to rozwiązać układ równań przy użyciu wzoru Shermana-Morrisona:

$$(B + uv^\mathsf{T})^{-1} = B^{-1} - \frac{B^{-}1uv^\mathsf{T}B^{-1}}{1 + v^\mathsf{T}B^{-1}u}$$

Dzięki zastosowaniu tego wzoru i postaci macierzy będziemy mogli rozwiązać układ równań $Au = b$ w czasie $O(N)$. Nasza macież B jest trójdiagonalna co pozwala rozwiązywać układy $y = B^{-1}b$ oraz $z = B^{-1}u$ w czasie $O(N)$ poprzez użycie algorytmu Thomasa.

5.2. Implementacja algorytmu Thomasa oraz rozwiązania wzorem Shermana-Morrisona

```
class ThomasSolver {
  Eigen::VectorXd y;
  Eigen::VectorXd z;
  Eigen::VectorXd c;
  /// Creates Thomas Solver for three diag matrix
  /// x upper diag 1 - N-1 (a_N is ignored)
  /// y diag 1 - N
  /// z lower diag 2 - N (c_1 is ignored)
public:
  // Based on: https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
  ThomasSolver(Eigen::VectorXd &&x, Eigen::VectorXd &&y, Eigen::VectorXd &&z)
      : y(std::move(y)), z(std::move(z)) {

    long N = this->y.size();
    this->c = Eigen::VectorXd(N);

    this->c(0) = x(0) / this->y(0);
    for (long i = 1; i < this->y.size(); i++) {
      this->c(i) = x(i) / (this->y(i) - this->z(i) * this->c(i - 1));
    }
  }

  Eigen::VectorXd solve(const Eigen::VectorXd &b) const {
    auto N = this->y.size();
    assert(b.size() == N);
    Eigen::VectorXd d = Eigen::VectorXd::Zero(N);

    d(0) = b(0) / y(0);
    // Calculate coefficients in forwoard sweep
    for (long i = 1; i < this->y.size(); i++) {
      d(i) = (b(i) - this->z(i) * d(i - 1)) /
             (this->y(i) - this->z(i) * this->c(i - 1));
    }

    auto x = Eigen::VectorXd(N);
    // Backsubstitute for x_n
    x(this->y.size() - 1) = d(N - 1);
    for (long i = N - 2; i >= 0; i--) {
      x(i) = d(i) - c(i) * x(i + 1);
    }

    return x;
  }
};
```

Kod 10: Własna implementacja algorytmu Thomasa

```
// Create specific solver for our B matrix (A = B + uuT)
// tr - top right corner
// bl - bottom left corner
ThomasSolver create_solver_for_B(long N, double tr, double bl) {
  Eigen::VectorXd a = Eigen::VectorXd::Zero(N);
  Eigen::VectorXd b = Eigen::VectorXd::Zero(N);
  Eigen::VectorXd c = Eigen::VectorXd::Zero(N);

  float64_t h = 2 / ((float64_t)N - 1.0);

  for (long n = 0; n < N; n++) {
    a(n) = 1.0 / (h * h);
    b(n) = -2.0 / (h * h);
    c(n) = 1.0 / (h * h);
  }
  // Fix B_(1,1) and B_(N, N)
  float64_t gamma = -b(0);
  b(0) -= gamma;
  b(N - 1) -= (tr * bl) / gamma;

  return ThomasSolver(std::move(a), std::move(b), std::move(c));
}
```

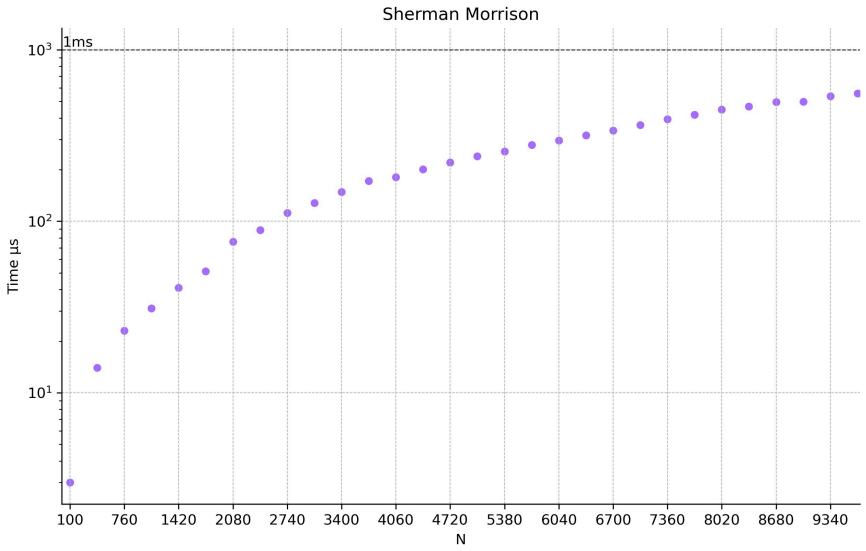
Kod 11: Solver dla macierzy B

```cpp
Eigen::VectorXd solve_mat_sherman_morrison(long N) {
  // Generate B matrix as threediagonal
  // Use thomas algorithm to solve intermidied equations for sherman morrison
  // solve for u using Sherman Morrison

  // A = B + uvT
  float64_t h = 2.0 / (N - 1);
  float64_t gamma = 2.0 / (h * h);

  // top left corner val
  float64_t tr = 1.0 / (h * h);
  // bototm lef corner val
  float64_t bl = 1.0 / (h * h);

  Eigen::VectorXd u = Eigen::VectorXd::Zero(N);
  u(0) = gamma;
  u(N - 1) = 1.0 / (h * h);
  Eigen::VectorXd v = Eigen::VectorXd::Zero(N);
  v(0) = 1;
  v(N - 1) = (bl * tr) / gamma;

  auto B = create_solver_for_B(N, tr, bl);

  auto b = gen_b_vector(N);
  // y = B^-1 b => By = b
  auto y = B.solve(b);
  // z = B^-1 u = Bz = u
  auto z = B.solve(u);

  float64_t p = (v.transpose() * z);
  Eigen::VectorXd x = y - (z * (v.transpose() * y)) / (1.0 + p);

  return x;
}
```

Kod 12: Rozwiązanie równania przy użyciu wzoru Shermana-Morrisona

5.3. Analiza rozwiazania

Wykorzystanie rozwiązania wykorzystującego specyfikę naszej macierzy jesteśmy w stanie znajdować rozwiązanie układu w czasie $O(N)$ (Tabela 7). W prównaniu dla rzadkiego rozkładu LU (Tabela 5) czasy rozwiązań są około 20 razy szybsze. Dla macierzy $N = 9670$ wykorzystanie alorytmu Thomasa i wzoru Shermana-Morrisona powoduje wykonanie obliczeń w czasie mniejszym niż $1ms$ co dla wcześniej omawianych metod mogło by zostać uznane jako wariacje czasów wykonania miedzy próbami.

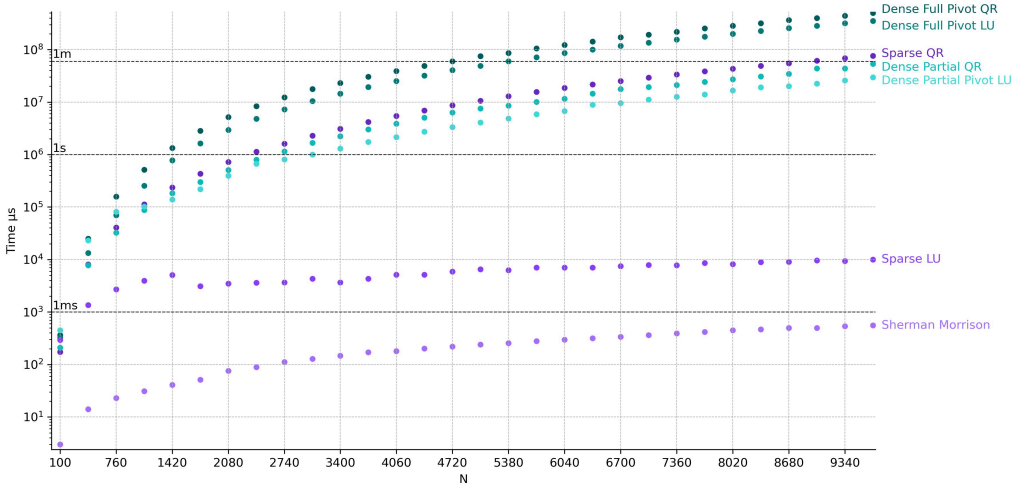


Wykres 9: Czas rozwiązania $Au = b$ dla różnych N przy użyciu wzoru Shermana-Morrisona i algorytmu Thomasa

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$3\mu s$	$31\mu s$	$76\mu s$	$128\mu s$	$181\mu s$	$240\mu s$	$296\mu s$	$365\mu s$	$448\mu s$	$498\mu s$	$557\mu s$

Tabela 7: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu wzoru Shermana-Morrisona i algorytmu Thomasa

6. Prównanie wszystkich metod

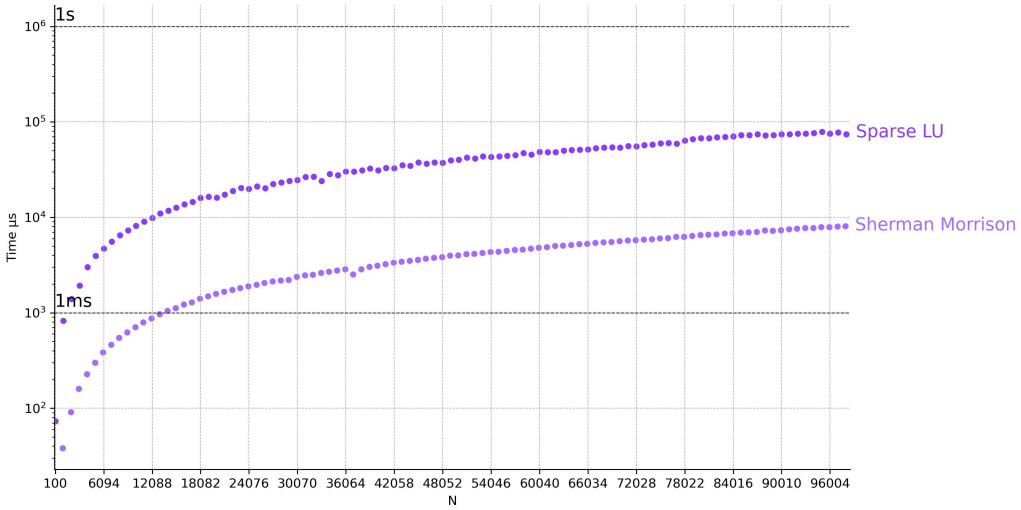


Wykres 10: Prównanie czasów rozwiązania $Au = b$ dla różnych N dla wszystkich omawianych metod

Na Wykres 10 widać zgrupowane metody, które układają się w cztery rozróżnialne grupy. Zaczynając od najwolniejszych są to metody używające rozkładów gęstych z pełnym pivotingiem. Są one o około 6 rzędów wielkości wolniejsze niż nasza najardziej efektywna metoda. Kolejną grupą są metody używające częściowego pivotingu oraz rozkład QR dla macierzy rzadkich. Są one o rząd wielkości szybsze niż metody używające pełengo pivotingu. Metody z tych dwóch grup chrakteryzują się czasem obliczeń $O(N^3)$ i róznią się jedynie stałym współczynnikiem niezależnym od wielkości macierzy. Pozostałe

dwie metody cechują się czasem obliczeń $O(N)$ i zostaną omówione dogłębniej w kolejnej sekcji, gdyż one jedyne są rozsądnym wyborem do szukania rozwiązań dla macierzy $N > 10000$.

6.1. Porównanie Sparse LU oraz wzoru Shermana-Morrisona z algorytmem Thomasa



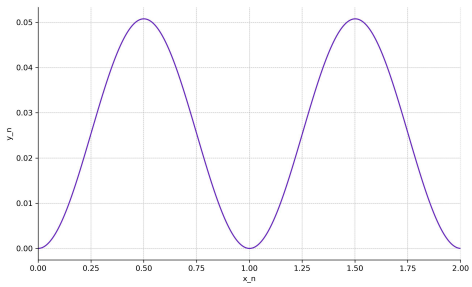
Wykres 11: Prównanie czasów rozwiązania $Au = b$ dla dużych N

N	1099	10090	20080	30070	40060	50050	60040	70030	80020	90010	99001
Sherman	$36\mu s$	$642\mu s$	$1426\mu s$	$2158\mu s$	$2446\mu s$	$3467\mu s$	$4169\mu s$	$4889\mu s$	$5609\mu s$	$6376\mu s$	$7452\mu s$
Sparse LU	$754\mu s$	$7630\mu s$	$15ms$	$23ms$	$30ms$	$37ms$	$44ms$	$47ms$	$59ms$	$63ms$	$68ms$

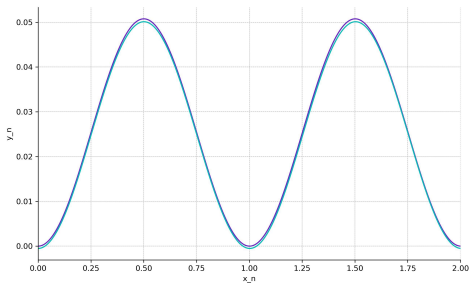
Tabela 8: Czasy rozwiązań $Au = b$ dla dużych N efektywnymi metodami

Analizując dane z Wykres 11 oraz Tabela 8 rozwiązywanie równania przy użyciu najbardziej optymalnej metody daje nam około 10-krotnie szybsze wykonanie obliczeń. Rzadki rozkład LU dla macierzy $N = 99001$ posiada porównywalny rząd czasu wykonania jak gęste LU z cześciowym pivotingiem (Tabela 2) lecz dla macierzy $N = 1090$, więc w porównywalym czasie jesteśmy w stanie policzyć macierz 98 razy większą. Ekstrapolując czas wykonania przy założeniu, że gęste LU z częściowym pivotem posiada złożoność obliczeniowa $O(N^3)$ policzenie macierzy $N = 96700$ zajeło by $29000s \simeq 8h$, pomijając fakt pamięci potrzebnej do jej przechowania (około 70GB).

6.2. Analiza wyników



Wykres 12: Wykres zależności u_n i wartości dla $N = 1000$



Wykres 13: Wykres przybliżenia wartości u_n dla $N = 1000$ przy użyciu $-\frac{\cos(6.27x)}{39.5} + 0.0248$

Wykres 12 pokazuje rozkład wartości wektora $\vec{u}$ unormowane na przedziale $(0, 2)$. Wartości te układują się w funkcje przypominająca $-\cos(x)$. Na Wykres 13 została nałożona przybliżenie stworzone przu użyciu funkcji $-\frac{\cos(6.27x)}{39.5} + 0.0248$. Jest to oczywiście przybliżenie, lecz można o użyć w metodzie iteracyjnej jako wektor początkowy co zagwarantuje nam szybką zbierzność i przyśpieszy czas rozwiązania równania. Głebsza analiza wyników i próba znalezienia dokładniejszego przybliżenia mogła by się okazać dokładnym rozwiązaniem naszego układu równań.

7. Podsumowanie

Po przeanalizowaniu wszystkich metod można zauważyć, że dobór odpowiedniej metody jest kluczową rzeczą przy rozwiązywaniu układów równań. Znajomość struktury macierzy pozwala nam skrócić czas wykonania o 6 rzędów wielkości dla średnich macierzy. Dla dużych macierzy różnice te są jeszcze bardziej widoczne i dobór i implementacja odpowiedniej metody jest bardziej efektywna czasowo niż siłowe rozwiązanie metodami gęstymi. (Opis czasów dla macierzy 100000x100000). Analiza wynikowego wektora może umożliwić nam znalezienie funkcji przybliżającej rozwiązanie. Może być użyteczne w przypadku macierzy, których unormowany rozkład wyników będzie posiadał podobną strukture, tak jak w naszym przypadku.