Metody Numeryczne N01

Jakub Kurek

1. Wstęp

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

$$A^{N\times N} = \begin{pmatrix} 1 & & & & \\ \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & & \\ & \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} \qquad h = \frac{2}{N-1} \qquad b = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

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

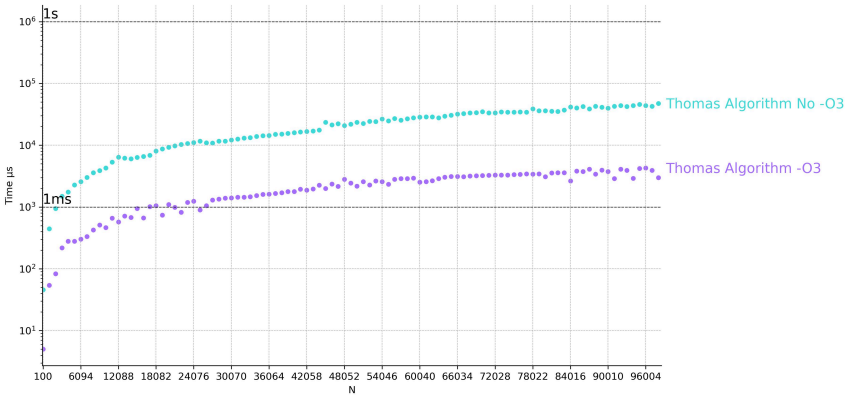
1.2. Metodyka testowania

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

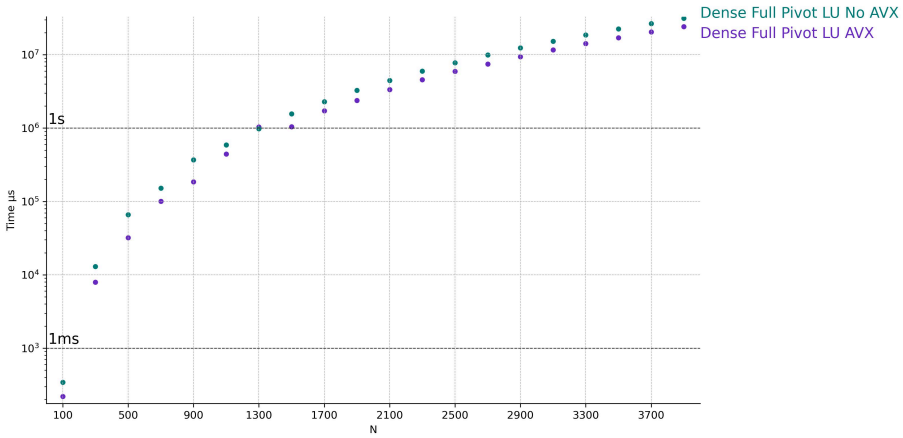
Najważniejszą optymalizacją poprawiające wyniki obliczeń było zastosowanie odpowiednich flag kompilatora:

- `-O3` - poziom optymalizacji kodu przez kompilator. Wykres 1 pokazuje róznice czasu wykonania zwiększoną około 10-krotnie.



Wykres 1: Porównanie algorytmu Thomasa 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.

```
Eigen::MatrixXd gen_dense_A(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) = 1.0;
  mat(N - 1, N - 1) = 1.0;

  return mat;
}
```

Kod 1: Generowanie gęstej macierzy pełnej

```
Eigen::VectorXd gen_b_vector(long N) {
  Eigen::VectorXd b = Eigen::VectorXd(N);
  b.setZero();
  b(0) = 1;
  b(N - 1) = 1;
  return b;
}
```

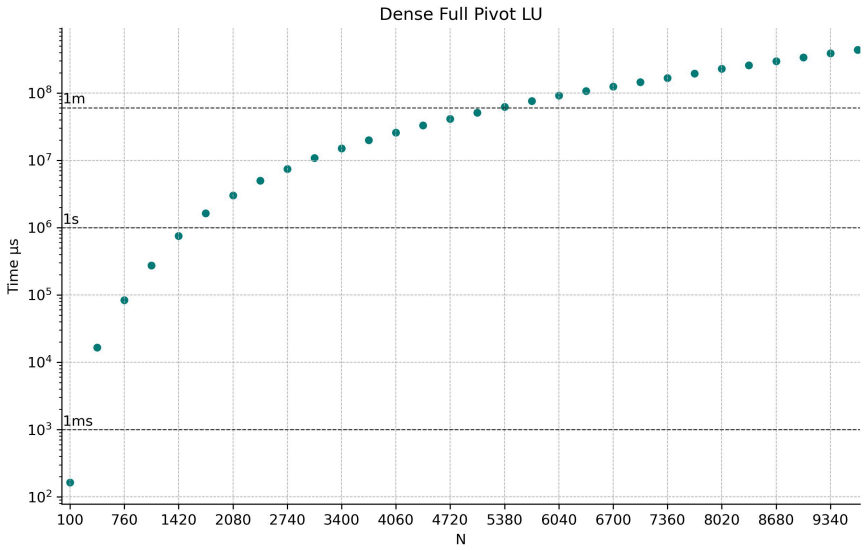
Kod 2: Generowanie wektora rozwiązania

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	$164\mu s$	$272ms$	$2998ms$	$10s$	$25s$	$50s$	$91s$	$144s$	$228s$	$336s$	$436s$

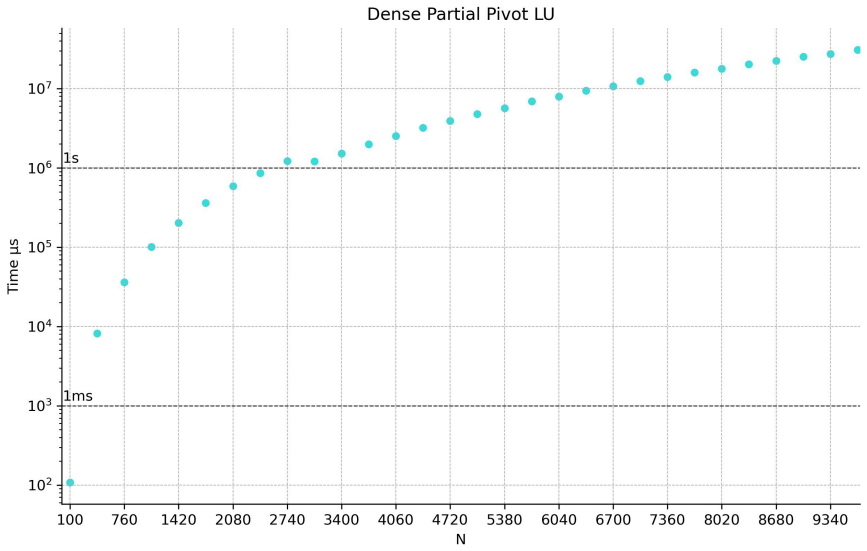
Tabela 1: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu fullPivLU

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 14-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	$108\mu s$	$100ms$	$591ms$	$1206ms$	$2530ms$	$4762ms$	$7967ms$	$12s$	$17s$	$25s$	$30s$

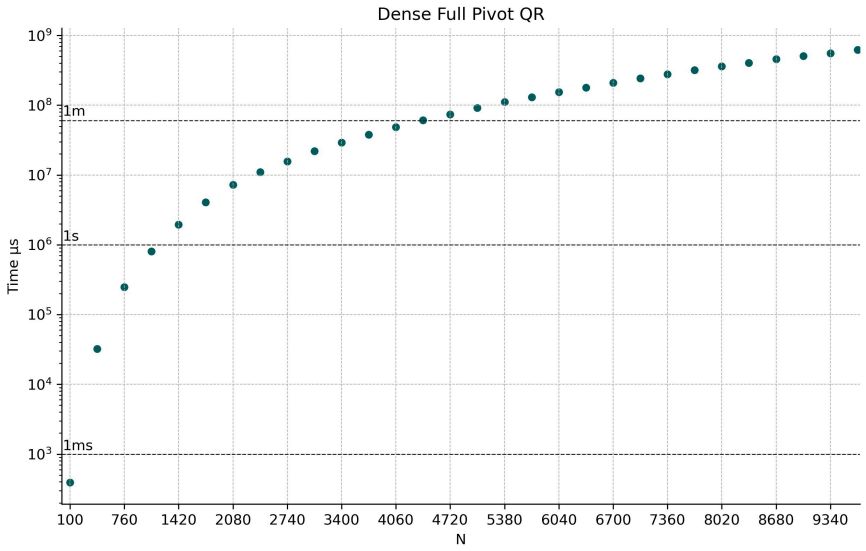
Tabela 2: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu partialPivLU

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	$393\mu s$	$807ms$	$7225ms$	$22s$	$48s$	$91s$	$153s$	$242s$	$360s$	$508s$	$621s$

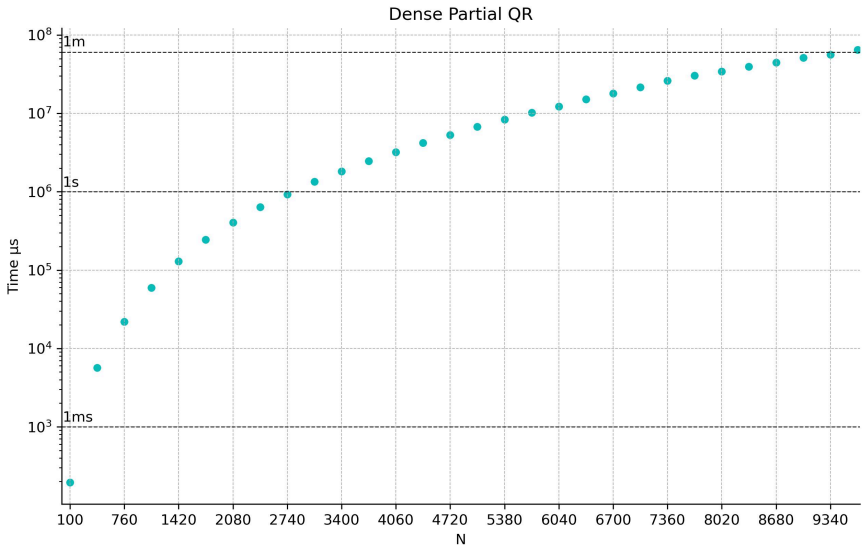
Tabela 3: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu fullPivHouseholderQr

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 2 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

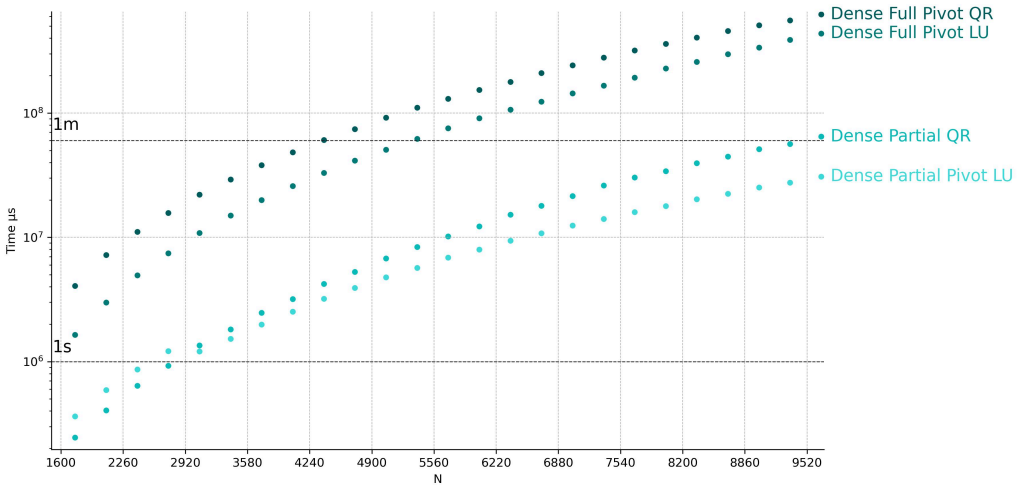


Wykres 6: Czas rozwiązania $Au = b$ dla różnych N przy użyciu householderQr

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$195\mu s$	$59ms$	$405ms$	$1349ms$	$3184ms$	$6780ms$	$12s$	$21s$	$34s$	$51s$	$64s$

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

3.5. Porównanie metod gęstych



Wykres 7: Czas rozwiązania $Au = b$ dla $2000 < N < 10000$ metody gęste

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 zmiana go na sparse vector nie oferuje zauważalnych zysków.

```
Eigen::SparseMatrix<double> gen_sparse_A(long N) {
  // matrix must be greater than 1
  // assert(N != 0);
  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 (int i = 1; i < N - 1; i++) {
    tripletList.push_back(Eigen::Triplet<double>(i, i, -2.0 / (h * h)));
    if (i < N - 1)
      tripletList.push_back(Eigen::Triplet<double>(i, i + 1, 1.0 / (h * h)));
    if (i > 0)
      tripletList.push_back(Eigen::Triplet<double>(i, i - 1, 1.0 / (h * h)));
  }
  sp_mat.setFromTriplets(tripletList.begin(), tripletList.end());

  sp_mat.coeffRef(0, 0) = 1;
  sp_mat.coeffRef(N - 1, N - 1) = 1;
  sp_mat.makeCompressed();
  return sp_mat;
}
```

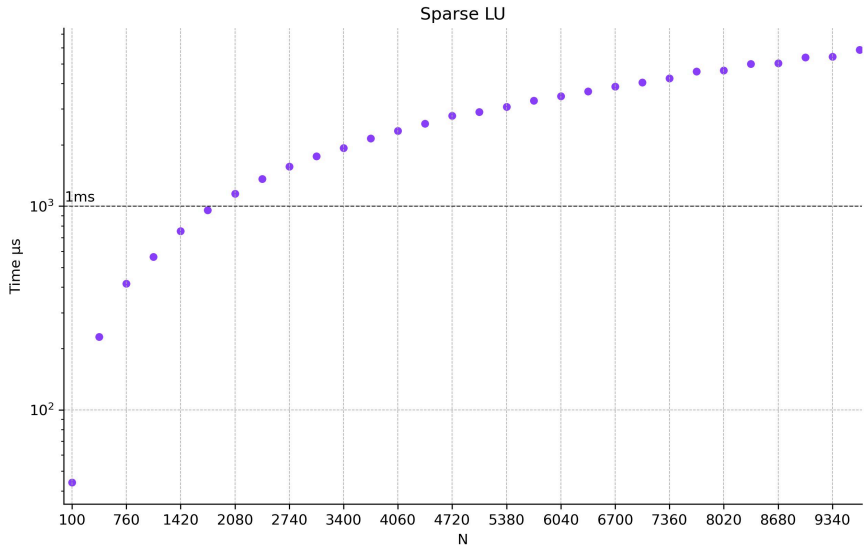
Kod 7: Generowanie rzadkiej macierzy

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 niewielki wpływ na efektywnośc metody posiada system operacyjny i przełączanie zadań (ostatnie pomiary nie są znacząco rosnące). 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 5000 razy.

```
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 8: Czas rozwiązania $Au = b$ dla różnych N przy użyciu sparse LU

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$44\mu s$	$563\mu s$	$1152\mu s$	$1756\mu s$	$2348\mu s$	$2902\mu s$	$3467\mu s$	$4057\mu s$	$4642\mu s$	$5378\mu s$	$5860\mu s$

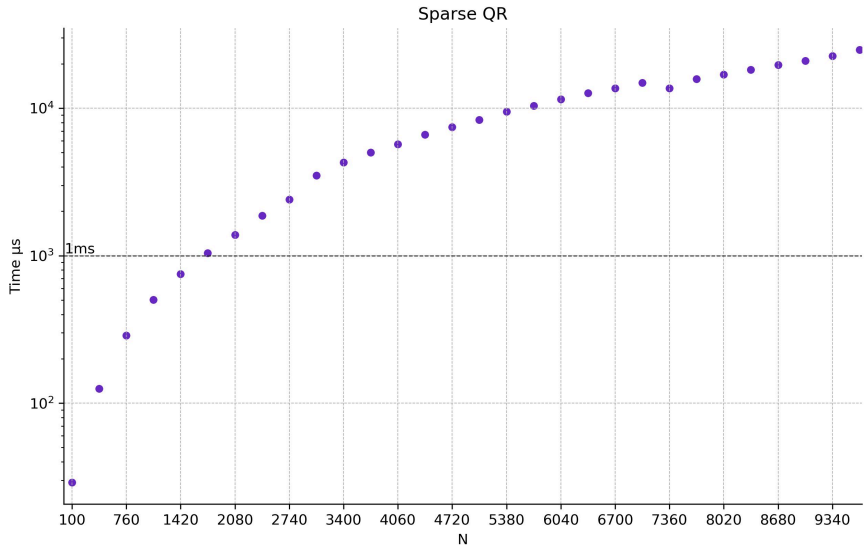
Tabela 5: Czasy rozwiązań $Au = b$ dla wybranych N przy użyciu sparse LU

4.2. Sparse QR

Drugą z oferowanych przez bibliotekę Eigen metod dekompozycji macierzy rzadkich jest zmodyfikowany rozkład QR. Analizując czasy z Tabela 5 zauważamy, że metoda jest około 5 razy wolniejsza niż odpowiadający jej rozkład LU. Z samych danych z tabeli ciężko stwierdzić złożonośc algorytmu, gdyż dla $N = 1090$ i $N = 2080$, dla 2-krotnego wzrostu, czas rośnie 2.5-krotnie. Natomiast dla macierzy $N = 4060$ i $N = 8020$ prawie 3-krotnie. Dokładniejsza analiza czasów wykonania i złożoności czasowej dla $N > 10000$ zostanie przeprowadzona wspólnie z resztą względnie efektywnych metod.

```
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 9: Czas rozwiązania $Au = b$ dla różnych N przy użyciu sparse QR

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$29\mu s$	$502\mu s$	$1385\mu s$	$3507\mu s$	$5687\mu s$	$8354\mu s$	$11ms$	$14ms$	$16ms$	$21ms$	$24ms$

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

5. Rozwiązanie wykorzystujące strukture macierzy

5.1. Obserwacje

Nasza macierz A posiada struktóre trójdagonalną co pozwala nam skorzystać z algorytmu Thomasa (brak zer na głównej diagonali), który sprowadza liczbę operacji do minimum i pozwala rozwiązać układ równań w czasie $O(N)$.

5.2. Implementacja algorytmu

```
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;
  }
};
```

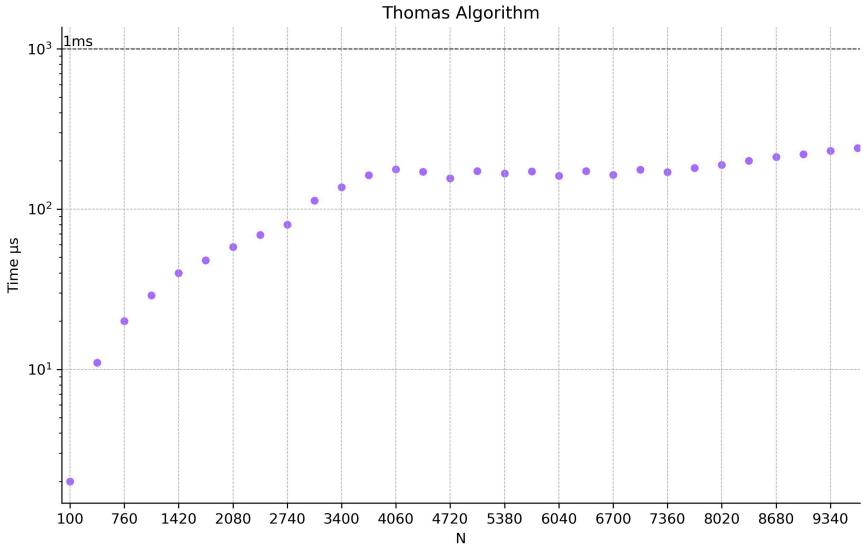
Kod 10: Własna implementacja algorytmu Thomasa

```
ThomasSolver create_solver_for_A(long N) {
  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);
  }
  a(0) = 0;
  c(N - 1) = 0;
  b(0) = 1;
  b(N - 1) = 1;

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

Kod 11: Solver dla macierzy A

5.3. Analiza wyników algorytmu Thomasa

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 krótsze. Dla macierzy $N = 9670$ wykorzystanie alorytmu Thomasa. Mimo powtarzania i uśredniania czasów rozwązań, dla macierzy $4000 < N < 8000$ osylują wokół jednego czasu, co wskazuje na duży wpływ systemu peracyjnego na rozwiazania, w tak krótkim czasie.

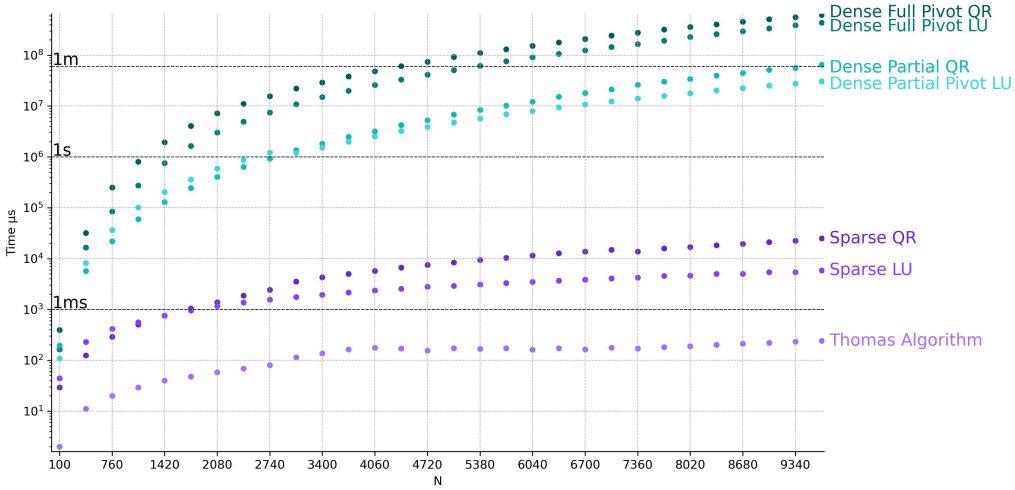


Wykres 10: Czas rozwiązania $Au = b$ dla różnych N przy algorytmu Thomasa

N	100	1090	2080	3070	4060	5050	6040	7030	8020	9010	9670
Czas	$2\mu s$	$29\mu s$	$58\mu s$	$113\mu s$	$177\mu s$	$173\mu s$	$161\mu s$	$176\mu s$	$189\mu s$	$220\mu s$	$241\mu s$

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

6. Prównanie wszystkich metod

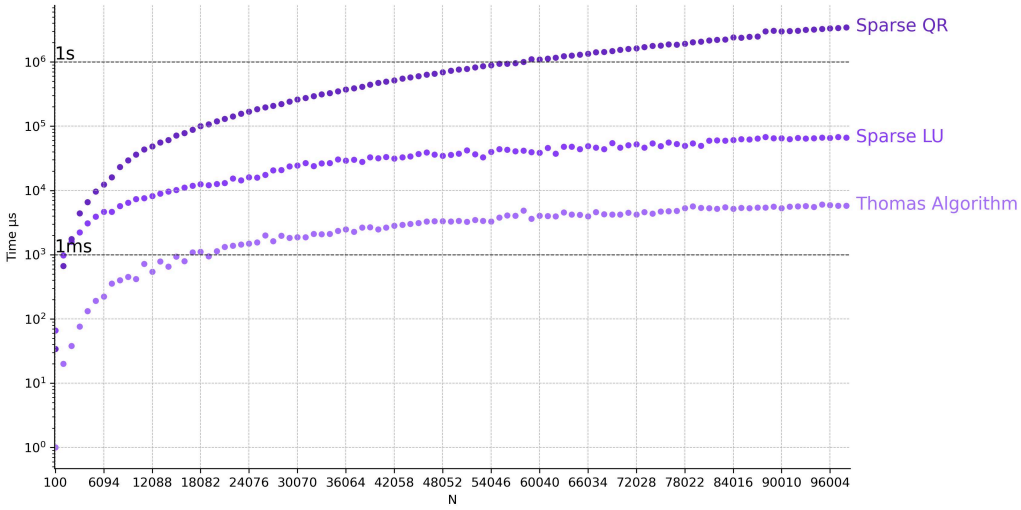


Wykres 11: Prównanie czasów rozwiązania $Au = b$ dla różnych N dla wszystkich omawianych metod

Na Wykres 11 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 algorytm Thomasa. Kolejną grupą są metody używające częściowego pivotingu, sa około 10-krotnie szybsze niż ich odpowiedniki z pełnym pivotingiem. Metody z tych dwóch grup chrakteryzują się czasem obliczeń $O(N^3)$ i róznią się jedynie stałym współczynnikiem niezależnym od wielkości macierzy. Nastepną grupą sa metody rzadkie. Najszybszy w osobnej grupie jest algorytm Thomasa. Metody z dwóch ostatnich grup zostaną omówione dogłebniej w kolejnej sekcji, gdyż one jedyne są rozsądnym wyborem do szukania rozwiązań dla macierzy $N > 10000$.

6.1. Porównanie metod efektywnych

Analizując dane z Wykres 12 dobrze widać, że nawet dla efektywnych metod czasy mogą się różnić o ponad 2-rzędy wielkosci (Alorytm Thomasa i Sparse QR). Dla rzadkiego rozkładu QR dopiero widać, iż nie rośnie on w czasie liniowym $O(N)$ tak jak rzadkie LU i algorytm Thomasa. Z danych Tabela 8 można zauwazyć, że złożoność czasowa rośnie w czasie wielomianowym $O(N^2)$. Dla pojedyńczych macierzy $N \simeq 100000$, wszystkie metody dają względnie rozsądne czasy rozwiązań, chociaż bezdyskusyjnie widać przewagę metod $O(N)$. Obie metody mimo liniowego czasu wykonania różnią się od siebie o rząd wielkości poprzez stały współczynik wykonywanyhch operacji.



Wykres 12: Prównanie czasów rozwiązania $Au = b$ dla dużych N

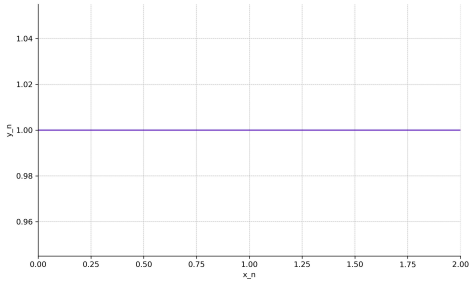
N	1099	10090	20080	30070	40060	50050	60040	70030	80020	90010	99001
Thomas	$20\mu s$	$420\mu s$	$1135\mu s$	$1878\mu s$	$2479\mu s$	$3369\mu s$	$4048\mu s$	$4233\mu s$	$5369\mu s$	$5328\mu s$	$6034\mu s$
Sparse LU	$977\mu s$	$7310\mu s$	$12ms$	$24ms$	$31ms$	$37ms$	$38ms$	$46ms$	$49ms$	$64ms$	$67ms$
Sparse QR	$671\mu s$	$36ms$	$119ms$	$261ms$	$472ms$	$767ms$	$1087ms$	$1540ms$	$2080ms$	$2986ms$	$3580ms$

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

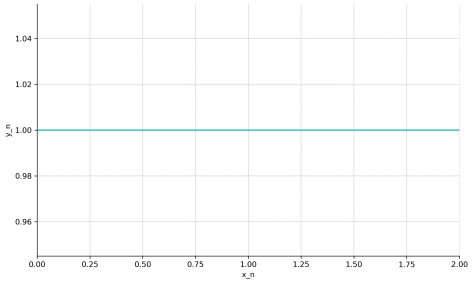
6.2. Analiza wyników

$$A^{N\times N} = \begin{pmatrix} 1 & & & & \\ \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & & \\ & \frac{1}{h^2} & \frac{-2}{h^2} & \frac{1}{h^2} & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

Analizując dokładniej strukturę naszej macierzy oraz wektora rozwiązania należy zwrócić uwagę na to, że u_1 oraz u_n muszą być równe 1. Dla $1 < n < N$ nasza macierz jest symetryczna i każdy wiersz musi się zerować. Elementy na lewo i lewo od diagonali po dodaniu towrzą element diagonalny z przeciwnym znakiem. Pierwsze równanie oraz ostatnie. Dyktują nam współczyniki przy u, które propagują się na cały wektor rozwiązania, który zapełnia się 1. Wykres 13 i Wykres 14 pokazują, że oczekiwania zgadzają się z rzeczywistością. Znając analizę tego wyniku, można zauważyć, że dla naszego b rozwiązanie u zawsze będzie mieć postać $u = \begin{pmatrix} 1 & \cdots & 1 \end{pmatrix}^\mathsf{T}$. Co neguje jakąkolwiek potrzebę przeprowadzania rozkładów macierzy A.



Wykres 13: Wykres zależności u_n i wartości dla $N = 1000$



Wykres 14: Wykres przybliżenia wartości u_n dla $N = 1000$ przy użyciu $u_n = 1$

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. Wstepna analiza rozwiązania pozwala nam, również zaoszczędzić czas przy liczeniu macierzy, lecz zyski czasowe będą nie wystarczające, gdyż wykorzystanie samejj postaci trójdiagonalnej będzie wystarczająca optymalizacją dla większości przypadków.