Thomas-algoritme en preconditionering

Arthur van Dam

November 2000

1 Inleiding

Het oplossen van niet al te grote lineaire stelsels geschiedt doorgaans met behulp van directe methoden. De stelsels die opgelost moeten worden, zijn van de vorm

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Een speciale matrix zal bestudeerd worden, namelijk een tridiagonale matrix $\mathbf{A}$. Tridiagonale matrices komen vaak voor bij driepunts eindige differenties en ook bij eindige elementen met lineaire interpolatie.

Een algoritme, afgeleid van Gauss-eliminatie met de modificatie van Crout, is het Thomas-algoritme.

2 Thomas algoritme

Stel dat $\mathbf{A}$ en $\mathbf{b}$ gegeven zijn door::

$$\mathbf{A} = \begin{pmatrix} \alpha_1 & \beta_1 & & & \varnothing \\ \gamma_2 & \alpha_2 & \beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \gamma_{n-1} & \alpha_{n-1} & \beta_{n-1} \\ \varnothing & & & \gamma_n & \alpha_n \end{pmatrix} \quad \text{en} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} \tag{1}$$

Het Thomas-algoritme maakt gebruik van deze tridiagonaal-structuur. Het algoritme verloopt in twee stappen. Eerst wordt $\mathbf{A}$ met een 'forward sweep' naar een rechterdriehoeksmatrix omgevormd door de elementen γ_i te elmineren en de elementen α_i te normeren op 1:

$$\mathbf{A}' = \begin{pmatrix} 1 & \beta'_1 & & & \varnothing \\ & 1 & \beta'_2 & & \\ & & \ddots & \ddots & \\ & & & 1 & \beta'_{n-1} \\ \varnothing & & & & 1 \end{pmatrix} \quad \text{en} \quad \mathbf{b}' = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_{n-1} \\ b'_n \end{pmatrix} \tag{2}$$

De nieuwe waarden van de elementen β'_i en b_i kunnen bepaald worden voor de eerste vergelijking:

$$\beta'_1 = \frac{\beta_1}{\alpha_1} \quad \text{en} \quad b'_1 = \frac{b_1}{\alpha_1}$$

en voor alle volgende vergelijkingen:

$$\beta'_i = \frac{\beta_i}{\alpha_i - \gamma_i \beta'_{i-1}} \quad \text{en} \quad b'_i = \frac{b_i - \gamma_i b'_{i-1}}{\alpha_i - \gamma_i \beta'_{i-1}}.$$

De tweede stap is terugsubstitutie, de 'backward sweep':

$$\begin{aligned} x_n &= b'_n \\ x_i &= b'_i - x_{i+1}\beta'_i \end{aligned}$$

Met $5n - 4$ delingen en vermenigvuldigingen is dit een zeer efficiënt algoritme. Er moet echter ook op de fout worden gelet. Een vereiste voor stabiliteit van het Thomas-algoritme is diagonaaldominantie van $\mathbf{A}$:

$$|\alpha_i| > |\beta_i| + |\gamma_i| \tag{3}$$

Dit, ter voorkoming van deling door kleine getallen. Aanvankelijk kleine afrondfouten op machineprecisie zouden dan namelijk in iedere volgende coëfficiënt sterker gaan wegen, waardoor de uiteindelijke oplossing grote fouten kan gaan bevatten.

3 Implementatie

Om verder onderzoek aan het Thomas-algoritme te doen, is het voorgaande geïmplementeerd in C++. In appendix A.1 staat de listing van het hoofdprogramma. In appendix A.2 staat de code voor het Thomas-algoritme. De overige appendices tonen listings van een aantal hulp-libraries die de auteur voor diverse programma's gebruikt.

Het programma kent een aantal opties, te weten:

`-v` - Verbose mode, de voortgang van het progamma wordt geprint.
`-s num` - De keuze van de testmatrix, 2 of 4, zie sectie 4.
`-n num` - De systeemgrootte n, alleen van toepassing op testmatrix 4.

Het programma initialiseert het systeem, afhankelijk van de opgegeven parameters, past het algoritme toe, en controleert de oplossing.

De controle van de oplossing wordt gedaan door de gevonden benadering $\tilde{\mathbf{x}}$ van de echte oplossing $\mathbf{x}$ te vermenigvuldigen met $\mathbf{A}$. Met de zo gevonden vector $\tilde{\mathbf{b}}$ wordt de maximale relatieve fout bepaald:

$$\varepsilon = \max_i \left| \frac{b_i - \tilde{b}_i}{b_i} \right|. \tag{4}$$

4 Experimenten

In navolging van de opgavennummering van *Opgave 2* van *NUST, 2000/2001*, worden er twee testmatrices 'mTwee' en 'mVier' beschouwd.

4.1 'mTwee', een kleine test

Als eerste wordt een klein stelsel beschouwd:

$$\mathbf{A} = \begin{pmatrix} 4 & 2 & 0 & 0 \\ 1 & 3 & 2 & 0 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 1 \end{pmatrix} \quad \text{en} \quad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix},$$

met als bekende oplossing $\mathbf{x}^T = (0.75 \ \ -1 \ 1.625 \ \ -0.625)$. Vooraf kan opgemerkt worden dat de eerste twee rijen (bijna) voldoen aan de voorwaarde (3). In de laatste twee rijen is de afwijking van de eis niet fors, dus de verwachting is dat het algoritme een goed resultaat oplevert.

Het Thomas-algoritme levert een oplossingsvector $\tilde{\mathbf{x}}^T = (0.75 \ \ -1 \ 1.625 \ \ -0.625)$, met een maximale relatieve fout $\varepsilon = 2.22045 \cdot 10^{-16}$. Deze fout is in de orde van machineprecisie en is dus niet te verbeteren.

4.2 'mVier', een slecht geconditioneerd systeem en de invloed van diagonaalpreconditionering.

Om uitgebreider naar de invloed van voorwaarde (3) te kijken, wordt het volgende systeem beschouwd:
A en **b** als in (1), met

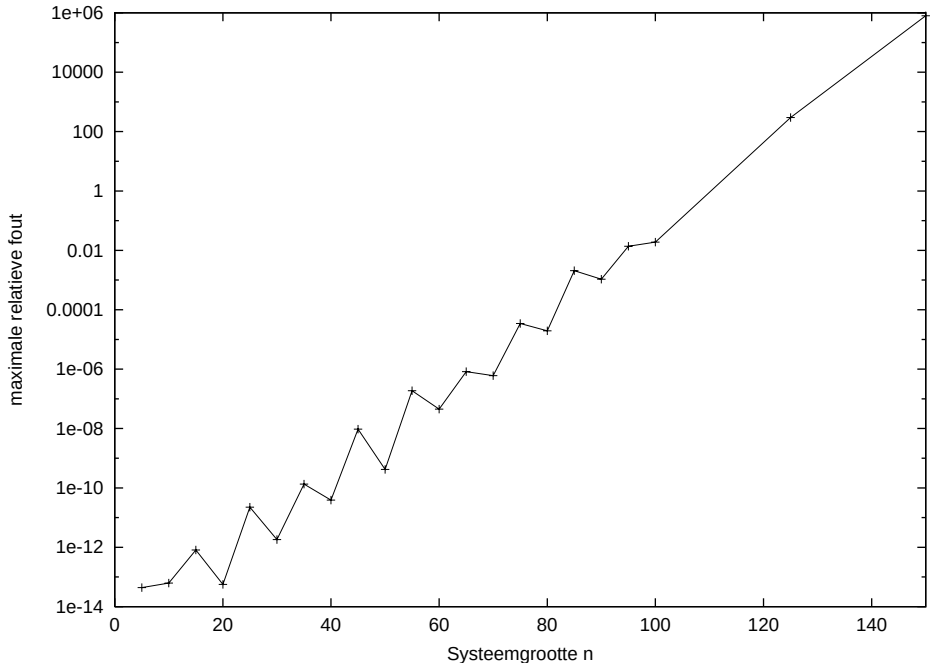
$$\begin{aligned} \alpha_i &= (n-i+1)\cdot 10^{-4} \\ \beta_i &= 2 \\ \gamma_i &= 1 \\ b_i &= 10^{-3} \end{aligned}$$

Voor geen enkele $n < 3 \cdot 10^{-4}$ wordt aan voorwaarde (3) voldaan. De resultaten, zoals getoond in tabel 1, maken duidelijk dat de afrondfouten hier inderdaad zwaar gaan doorwerken. Zelfs bij het kleine systeem $n = 5$ is de fout al 50 maal zo groot als de fout bij het testprobleem 'mTwee'.

Tabel 1: Maximale relatieve fouten voor 'mVier' en de gepreconditioneerde versie, voor diverse systeemgrootten.

n	ε bij 'mVier'	ε bij 'mVier' met preconditionering
5	4.40186e-14	4.40536e-14
10	6.26669e-14	1.5099e-14
15	8.09032e-13	8.0933e-13
20	5.61617e-14	7.79377e-15
25	2.25416e-11	2.25417e-11
30	1.82233e-12	4.61714e-13
35	1.33842e-10	1.33841e-10
40	3.8925e-11	3.89069e-11
45	9.50506e-09	9.50644e-09
50	4.19067e-10	4.19252e-10
55	1.87993e-07	1.87974e-07
60	4.48758e-08	1.33312e-08
65	8.26593e-07	8.26642e-07
70	6.01768e-07	6.01947e-07
75	3.44769e-05	2.51498e-05
80	1.94866e-05	1.94976e-05
85	0.00207771	0.00207726
90	0.00107225	0.00107241
95	0.0137521	0.0137582
100	0.0189895	0.0190112
125	296.607	347.469
150	797001	797211
200	2.91588e+13	2.91452e+13
500	1.39074e+59	2.50827e+59
1000	3.35493e+134	3.36265e+134

Voor systemen met dimensie $n < 150$ zijn de resultaten in figuur 1 getoond. Het globale verloop van de lijn lijkt lineair. De logschaal op de ε-as suggereert hiermee een exponentiële groei van de maximale relatieve fout.



Figuur 1: Maximale relatieve fout voor het ongepreconditioneerde stelsel 'mVier'

4.2.1 Diagonaalpreconditionering

Bij iteratieve methoden wordt vaak een preconditionering met een matrix $\mathbf{H}$ toegepast:

$$\mathbf{H}^{-1}\mathbf{A}\mathbf{x} = \mathbf{H}^{-1}\mathbf{b},$$

voor betere convergentie. Een simpele preconditioneerder, die toch al de nodige verbetering brengt, is $\mathbf{H} = \text{diag}(\mathbf{A})$.

Bij directe methoden echter, heeft dit minder effect. In het geval van de diagonaalpreconditioneerder, wordt puur iedere rij i met een factor $1/\alpha_i$ herschaald. Dit verandert dus niks aan de voorwaarde (3). De derde kolom in tabel 1 toont inderdaad aan, dat geen betere resultaten geboekt worden.

Eventueel zou verder onderzoek gedaan kunnen worden naar een preconditionering die de drie diagonaalelementen dusdanig herschaald, dat wel aan de voorwaarde (3) is voldaan.

5 Conclusie

Het Thomas-algoritme blijkt inderdaad zeer snel te zijn. Er zijn weliswaar geen tijdmetingen uitgevoerd, maar alle experimenten konden 'on the fly' uitgevoerd worden.

De nauwkeurigheid blijk sterk samen te hangen met de voorwaarde (3). Indien hier niet aan is voldaan, wordt zelfs voor kleine systemen een fout fors groter dan de machineprecisie gemaakt.

Zoals verwacht levert de diagonaalpreconditionering geen verbetering op.

Referenties

[1] .J. van Leeuwen, Numerieke Stromingsleer, Utrecht

A Listings

Listing A.1 bevat het hoofdprogramma. In de `solvers`-library (listing A.2) staat de `ThomasSolve`-routine. De rest van de libraries zijn aanvullende tools.

A.1 `problem2.cc`

Het hoofdprogramma bij deze opgave.

```
#include "blas.h"
#include "solvers.h"
#include "printers.h"
#include "alloc.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/**
 * core-module for task 2, NUST 2000/2001
 * note that indices start at 0 in code below, whilst they start at 1 in task.
 *
 * author: Arthur van Dam
 */
bool VERBOSE = false;

/**
 * testmatrix from sub-task 2
 */
void initTwo(unsigned int n, double *alpha, double *beta, double *gamma, double *b)
{
  alpha[0] = 4.0;
  alpha[1] = 3.0;
  alpha[2] = 2.0;
  alpha[3] = 1.0;
  beta[0] = 2.0;
  beta[1] = 2.0;
  beta[2] = 2.0;
  gamma[1] = 1.0;
  gamma[2] = 1.0;
  gamma[3] = 1.0;
  b[0] = 1.0;
  b[1] = 1.0;
  b[2] = 1.0;
  b[3] = 1.0;
}

/**
 * testmatrix from sub-task 4
 */
void initFour(unsigned int n, double *alpha, double *beta, double *gamma, double *b)
{
  int i, ri;

  for(i = 0; i < n; i++){
    alpha[i] = (n-i)*1.e-4; // omit +1, because i starts at 0
    beta[i] = 2.0;
    gamma[i] = 1.0;
    b[i] = 1.e-3;
  }
}

double checkError(unsigned int n, double *alpha, double *beta, double *gamma, double *b, double *x)
{
  double *b_tilde;
  double *res;
  double max_res;

  b_tilde = vecallocd(n);
  res = vecallocd(n);

  tridiagMV(n, alpha, beta, gamma, x, b_tilde); // determine b_tilde, by substituting x

  if(VERBOSE)
  { printf("b_tilde=");
    printVector(n, b_tilde, VECMODE_VERTICAL);
    printf("\n");
  }
```

```
  axpy(n, -1.0, b_tilde, b, res);            // ERROR

  rel_vec(n, res, b, res);                   // RELATIVE error

  if(VERBOSE)
  { printf("(b-b_tilde)/b=");
    printVector(n, res, VECMODE_VERTICAL);
    printf("\n");
  }

  max_res = absmax(n, res);                  //MAXIMAL relative error
  if(VERBOSE)
  { printf("  Maximal relative error: %g\n", max_res);
  }
  return max_res;
}

int main(int argc, char *argv[])
{
  int n = 0, mtxType = 0;
  double *alpha, *beta, *gamma, *b, *x;
  double eps;


/* Handle commandline arguments */
  for(int ix = 1; ix < argc; ix++)
  { char *pchar = argv[ix];
    switch ( pchar[0] )
    {
      case '-':
      {
        switch( pchar[1] )
        { case 'v':
            VERBOSE = true;
            break;
          case 's':
            mtxType = atoi(argv[++ix]);
            break;
          case 'n':
            n = atoi(argv[++ix]);
            break;
        }
      }
      break;
    }
  }
/** end parameter-processing **/

  if(mtxType == 0)
  { fprintf(stderr, "ERROR: no systemtype specified, use '-s [type(=2|4)]'\n");
    exit(1);
  }
  if(mtxType == 4 && n == 0)
  { fprintf(stderr, "ERROR: no systemsize specified, use '-n [systemsize]'\n");
    exit(1);
  }

  if(mtxType == 2)
  { n = 4;
  }

  if(VERBOSE)
  { printf("+=================================================================+\n");
    printf("|  NUST problem 2: Thomas algorithm and preconditioning         |\n");
    printf("+-----------------------------------------------------------------+\n");
  }


  /* Allocate data-structures */
  alpha = vecallocd(n);
  beta = vecallocd(n);
  gamma = vecallocd(n);
  b = vecallocd(n);
  x = vecallocd(n);

  /* initialize data structures */
  switch(mtxType)
  { case 2 :
      initTwo(n, alpha, beta, gamma, b);
      if(VERBOSE)
```

```
      {  printf("  Using matrix from (2) with matrixsize %d.\n", n);
      }
      break;
    case 4 :
      initFour(n, alpha, beta, gamma, b);
      if(VERBOSE)
      { printf("  Using matrix from (4) with matrixsize %d.\n", n);
      }
      break;
    default :
      fprintf(stderr, "Invalid systemtype (%d), exiting.\n", mtxType);
      exit(1);
      break;
  }

  /* solve system */
  if(VERBOSE)
  { printf("  Solving system...\n\n");
  }
  ThomasSolve(n, alpha, beta, gamma, b, x);

  /* check error */
  if(VERBOSE)
  { printf("  Checking error...\n\n");
  }
  eps = checkError(n, alpha, beta, gamma, b, x);
  if(!VERBOSE)
  { printf("%d %g", n, eps);
  }

  if(VERBOSE)
  { printf("solution-vector x=");
    printVector(n, x, VECMODE_HORIZONTAL);
    printf("\n");
  }


/* Now preconditioning with H=D */
  if(VERBOSE)
  { printf("+----------------------------------------------------------------+\n");
    printf(" Preconditioning with H=D\n  \nSolving system...\n\n");
  }
  rel_vec(n, beta, alpha, beta);
  rel_vec(n, gamma, alpha, gamma);
  rel_vec(n, b, alpha, b);
  rel_vec(n, alpha, alpha, alpha);
  ThomasSolve(n, alpha, beta, gamma, b, x);


  if(VERBOSE)
  { printf("  Checking error...\n\n");
  }
  eps = checkError(n, alpha, beta, gamma, b, x);
  if(!VERBOSE)
  { printf(" %g\n", eps);
  }

  if(VERBOSE)
  { printf("solution-vector x=");
    printVector(n, x, VECMODE_HORIZONTAL);
    printf("\n");
  }

  free(alpha);
  free(beta);
  free(gamma);
  free(b);
  free(x);

  if(VERBOSE)
  { printf("+================================================================+\n");
  }
  return 0;
}
```

A.2 `solvers.c`

Bevat het Thomas-algoritme.

```
#include <stdlib.h>
#include "alloc.h"

/**
 * solvers.c
 * a collection of solving routines for matrix-vector systems
 *
 * @author Arthur van Dam (adam@cs.uu.nl)
 * @lastmodified 7 november 2000
 */

/**
 * solves Ax=b in which A is a tridiagonal matrix
 *
 * @param n the system-size n
 * @param alpha the middle diagonal (length n)
 * @param beta the upper diagonal (length n-1)
 * @param gamma the lower diagonal (length n-1)
 * @param b the vector b
 * @param x the solution vector x into which the solution-data will be written
 */
void ThomasSolve(unsigned long n, double *alpha, double *beta, double*gamma, double *b, double *x)
{
  int i;
  double fraction;
  double *beta_acc;
  double *b_acc;

  beta_acc = vecallocd(n);
  b_acc    = vecallocd(n);

  beta_acc[0] = beta[0]/alpha[0];
  b_acc[0] = b[0]/alpha[0];

  for(i = 1; i < n; i++){
    fraction = 1.0 / (alpha[i] - gamma[i]*beta_acc[i-1]);
    beta_acc[i] = beta[i] * fraction;
    b_acc[i] = (b[i]-gamma[i]*b_acc[i-1]) * fraction;
  }

  x[n-1] = b_acc[n-1];

  for(i = n-2; i >= 0; i--){
    x[i] = b_acc[i] - x[i+1]*beta_acc[i];
  }
  free(beta_acc);
  free(b_acc);
}
```

A.3 alloc.c

Alloceert geheugen voor diverse datastructuren.

```
#include "stdio.h"
#include <stdlib.h>

/**
 * alloc.c
 * a toolset for allocating memory for various datastructures.
 *
 * @author Arthur van Dam (adam@cs.uu.nl)
 * @lastmodified 10 september 2000
 */

double *vecallocd(int n){
    /* This allocates a vector of doubles of length n */
    double *pd;

    pd= (double *)calloc(n,sizeof(double));
    if (pd==NULL)  exit(-1);
    return pd;          }

char *vecallocc(int n){
    /* This allocates a vector of chars of length n */
    char *pc;

    pc= (char *)calloc(n,sizeof(char));
    if (pc==NULL)  exit(-1);
    return pc;          }
```

```
int *vecalloci(int n){
    /* This allocates a vector of integers of length n */
    int *pi;

    pi= (int *)calloc(n,sizeof(int));
    if (pi==NULL)  exit(-1);
    return pi;       }


char **matallocc(int m, int n){
    /* This allocates an m x n matrix of chars */
    int i;
    char **ppc;

    ppc= (char **)calloc(m,sizeof(char *));
    if (ppc==NULL)   exit(-1);
    for (i=0; i<m; i++){
       ppc[i]= (char *)calloc(n,sizeof(char));
       if (ppc[i]==NULL)  exit(-1);  }
    return ppc; }

double **matallocd(int m, int n){
    /* This allocates an m x n matrix of doubles */
    int i;
    double **ppd;

    ppd= (double **)calloc(m,sizeof(double *));
    if (ppd==NULL)   exit(-1);
    for (i=0; i<m; i++){
       ppd[i]= (double *)calloc(n,sizeof(double));
       if (ppd[i]==NULL)  exit(-1);  }
    return ppd; }


int **matalloci(int m, int n){
    /* This allocates an m x n matrix of integers */
    int i;
    int **ppi;

    ppi= (int **)calloc(m,sizeof(int *));
    if (ppi==NULL)  exit(-1);
    for (i=0; i<m; i++){
        ppi[i]= (int *)calloc(n,sizeof(int));
        if (ppi[i]==NULL)  exit(-1);  }
    return ppi; }
```

A.4 printers.c

Voor het overzichtelijk uitprinten van datastructuren.

```
#include <stdio.h>
#include "printers.h"

/**
 * printers.c
 * a collection of solving routines for matrix-vector systems
 *
 * @author Arthur van Dam (adam@cs.uu.nl)
 * @lastmodified 7 november 2000
 */

/**
 * prints a vector to standard output
 *
 * @param n the length of the vector
 * @param x the vector to be printed
 * @param vecmode indicates whether to plot
 *    all elements as one row (VECMODE_HORIZONTAL)
 * or one element per row (VECMODE_VERTICAL)
 */
void printVector(unsigned int n, double *x, unsigned int vecmode)
{
  int i;
  printf("[\n");
  for(i = 0; i < n; i++){
    printf("%g", x[i]);
    if((vecmode & VECMODE_HORIZONTAL) == VECMODE_HORIZONTAL)
    { if(i != n-1)
```

```
      { printf(" ");
      }
    }
    else if((vecmode & VECMODE_VERTICAL) == VECMODE_VERTICAL)
    { printf("\n");
    }
  }
  printf("]");
}
```

A.5 blas.c

Een aantal BLAS-functies.

```
/**
 * blas.c
 * a collection of BLAS routines together with some matrix-routines.
 *
 * @author Arthur van Dam (adam@cs.uu.nl)
 * @lastmodified 18 september 2000
 */

void tridiagMV(unsigned int n, double *alpha, double *beta, double *gamma, double *x, double *b)
{
  int i;

  b[0] = alpha[0]*x[0] + beta[0]*x[1];
  for(i = 1; i < n-1; i++){
    b[i] = alpha[i]*x[i];
    b[i] += (gamma[i]*x[i-1] + beta[i]*x[i+1]);
  }
  b[n-1] = gamma[n-1]*x[n-2] + alpha[n-1]*x[n-1];
}

void axpy(unsigned long n, double alpha, double *x, double *y, double *z)
{ /* z=alpha*x+y */
  int i;
  if(alpha == 0.0){
    for(i = 0; i < n; i++){
      z[i] = y[i];
    }
    return;
  }
  if(alpha == 1.0){
    for(i = 0; i < n; i++){
      z[i] = x[i] + y[i];
    }
    return;
  }
  if(alpha == -1.0){
    for(i = 0; i < n; i++){
      z[i] = y[i] - x[i];
    }
    return;
  }

  for(i = 0; i < n; i++){
    z[i] = alpha*x[i] + y[i];
  }
  return;
}

void rel_vec(unsigned long n, double *x, double *y, double *z)
{ /* z[i] = x[i]/y[i] */
  int i;
  for(i = 0; i < n; i++){
    z[i] = x[i]/y[i];
  }
  return;
}

double absmax(unsigned int n, double *x){
  int i;
  double abs_max = 0.0;

  for(i = 0; i < n; i++){
    if(fabs(x[i]) > abs_max){
      abs_max = fabs(x[i]);
    }
```

```
  }
  return abs_max;
}
```