# Приложен софтуер, инсталиран на BG/P, част II

Явор Вутов

Институт по Информационни и Комуникационни Технологии
Българска Академия на Науките

24 Март 2010, София

# Съдържание

# I. Част
# Обща информация

# HDF5

## HDF5

Инсталирана е версия 1.8.4 на HDF5. HDF5 е модел за представяне на данни, файлов формат, както и библиотека за четене, запис и обработка на данни.
HDF5 дава възможност за гъвкав и ефективен достъп до големи обеми от различни видове данни. Библиотеката е преносима и разширяема и позволява на програмите гладко преминаване към използването и. Включени са и инструменти за обработка, разглеждане и анализиране на данни във формата HDF5.

# ParMetis

## ParMetis

ParMETIS е MPI базирана паралелна библиотека, която съдържа множество алгоритми за разделяне на неструктурирани графи и за пренареждане на елементите на разредени матрици. ParMETIS е особено подходящ за паралелни числени симулации, включващи големи неструктурирани мрежи. За такъв тип изчисления ParMETIS драстично намалява времето за комуникация чрез разделяне на мрежата така, че броят на елементите по интерфейса са минимизирани.

# ParMetis

## ParMetis

- Разделяне на неструктурирани графи и мрежи.
- Повторно разделяне на графи, които отговарят на адаптивно рафинирани мрежи.
- Разделяне на графи за multi-phase и multi-physics симулации.
- Подобряване на качеството на съществуващи разделяния.
- Изчислява fill-reducing orderings за разредена директна факторизация.
- Конструира дуални графи на мрежи.

# Trilinos

## Trilinos

Проектът Trilinos разработва робастни алгоритми за научни и инженерни приложения върху паралелни компютри и прави тези алгоритми достъпни за разработчици на приложения по най-ефективен начин.

## Trilinos

Trilinos съдържа: Pliris - Паралелна LU декомпозиция за пълни матрици, AztecOO - итерационни методи (CG, GMRES, BiCGStab), Belos - общи итерационни методи независещи представянето на векторите и матриците, Komplex - за решаване на системи от комплексни числа, Amesos - директни методи за решаване на системи.

# Trilinos

## Trilinos

Trilinos включва още и алгоритми в следните области:

- автоматично диференциране
- разпределение на данни за балансирано натоварване и робастност
- многонивови преобусловители
- блочни итерационни методи (линейни солвери)
- непълни факторизации
- решаване на линейни системи с последователни и едновременни десни части
- нелинейни методи
- оптимизация за задачи с голяма и много голяма размерност

# Trilinos

## Преобусловители

- AztecOO — преобусловители от тип непълна факторизация
- IFPACK — паралелни алгебрични преобусловители базирани на разделяне на областта с припокриване
- ML — многонивови паралелни алгебрични преобусловители
- Meros — паралелни блочни преобусловители

# HYPRE

## HYPRE

HYPRE е софтуерна библиотека, съдържаща високо производителни преобусловители и методи за решаване на големи, разредени системи линейни уравнения на паралелни компютри. Библиотеката предоставя паралелни multigrid методи за решаване на задачи както върху структурирани, така и върху неструктурирани мрежи.

## HYPRE

Съдържа методите на спреднатия градиент (CG), генерализиран метод на минималния резсидул (GMRES), стаблизиран биортогонализиран метод на спрегнатия градиент(BiCGStab), реализация на алгебричен мултигрид (BoomerAMG).

# HYPRE

## Преобусловители

- Jacobi
- BoomerAMG — паралелен мултигрид
- Euclid — паралелна непълна факторизация
- PILUT — паралелна непълна факторизация
- ParaSails — паралелен преобусловител на базата на приближена разредена обратна матрица

# II. Част
# Някои примери за употреба

# ParMETIS

## Разпределяне на мрежата между процесорите

Използваме ParMETIS за целта. Извикваме функцията

```
ParMETIS_V3_PartMeshKway (
  idxtype *elmdist, idxtype *eptr, idxtype *eind,
  idxtype *elmwgt, int *wgtflag, int *numflag,
  int *ncon, int *ncommonnodes, int *nparts,
  float *tpwgts, float *ubvec, int *options,
  int *edgecut, idxtype *part, MPI_Comm *comm)
```

# ParMETIS

## Параметри

Параметрите **elmdist**, **eptr** и **eind** задават елементите. Преди да извикаме ParMETIS, трябва някак (както и да е) да сме разпределили елементите по процесорите. Най добре да ги разделиме (приблизително) по равно. Параметърът **elmdist** съдържа началното разпределение на елементите по процесорите (къде колко има), **eptr** съдържа началните индекси на всеки елемент, а в **eind** са върховете, които образуват всички елементи. Параметрите **elmwgt** и **wgtflag** задават тежести по елементите (къде и по колко), **numflag** задава номерацията на възлите, дали започва от 0 или 1

# ParMETIS

## Параметри

Параметърът **ncommonnodes** определя свързаността на графа (за тримерни тетраедрални мрежи слагаме 3 — елементите са съседни, ако имат 3 общи върха. Броят на желаните части слагаме в **nparts** — ние искаме частите да са колкото и процесорите на които решаваме.
С параметрите **ncon**, **tpwgt** и **ubvec** задаваме тежести на елементите, например ако искаме да разделяме на области с различна големина — задаваме различни тежести (в нашият случай слагаме равни тежести за всички области).

# ParMETIS

## Параметри

В параметъра **edgescut** получаваме броя на разрезите м/у елементите, а в **part** получаваме за всеки елемент, номера на процесора в който трябва да го сложим. Освен тези се задават и параметрите **options** с допълнителни настройки на разделянето и **comm** в който задаваме комуникатора на MPI.

# Hypre

## Дискретизация

С функциите `HYPRE_IJVectorCreate`, `HYPRE_IJVectorSetObjectType` и `HYPRE_IJVectorInitialize` създаваме необходимите ни вектори (за дясната част и за решението), а с функциите `HYPRE_IJMatrixCreate`, `HYPRE_IJMatrixSetObjectType` и `HYPRE_IJMatrixInitialize` създаваме матрицата на системата линейни уравнения. След това обхождаме всички локални елементи, смятаме елементните матрици на коравина и десни част и с функциите `HYPRE_IJMatrixAddToValues` `HYPRE_IJVectorAddToValues` ги добавяме към глобалните матрица и вектор на дясната страна. Накрая приключваме асемблирането с извикване на функциите `HYPRE_IJMatrixAssemble` и `HYPRE_IJVectorAssemble`.

# Hypre

## Решаваме

```
//създаваме солвъра
HYPRE_ParCSRPCGCreate(comm, &PCGsolver);
//задаваме желаната грешка
HYPRE_PCGSetTol(PCGsolver, 1e-9);
//задаваме желания критерий за край
HYPRE_ParCSRPCGSetRelChange(PCGsolver, 1);
//задаваме максимален брой на итерациите
HYPRE_PCGSetMaxIter(PCGsolver, 100);
//създаваме преобусловителя
HYPRE_BoomerAMGCreate(&solver);
//подсказваме му, че ни е симетрична матрицата
HYPRE_BoomerAMGSetSym(solver, 1);
```

# Hypre

## Решаваме

```
//искаме да ни се докладва какво се случва
HYPRE_BoomerAMGSetPrintLevel(solver,1);
HYPRE_PCGSetPrintLevel(PCGsolver, 3);
//максимален брой итерации за преобусловителя
HYPRE_BoomerAMGSetMaxIter(solver, 1);
//максимален брой нива на мултигрида
HYPRE_BoomerAMGSetMaxLevels(solver, 25);
//задаване на някои параметри за мултигрида
HYPRE_BoomerAMGSetCoarsenType(solver, 8);
HYPRE_BoomerAMGSetRelaxType(solver, 6);
HYPRE_BoomerAMGSetTol(solver, 0);
```

# Hypre

## Решаваме

```
//задаване на преобусловителя
HYPRE_PCGSetPrecond(PCGsolver,
  (HYPRE_PtrToSolverFcn) HYPRE_BoomerAMGSolve,
  (HYPRE_PtrToSolverFcn) HYPRE_BoomerAMGSetup,
  solver);
//конструиране на преобусловителя
HYPRE_ParCSRPCGSetup(PCGsolver, mat, vec, x);
//решаваме задачата!
HYPRE_ParCSRPCGSolve(PCGsolver, mat, vec, x);
```

# Работа с вектори

## Пакетите Epetra и EpetraExt

Векторите са фундаментални структури от данни използвани от почти всички числени методи. В средата Trilinos, векторите се конструират посредством класове от Epetra.

В Epetra векторите могат да съхраняват или стойност с плаваща точка с двойна точност (например решение на ЧДУ или дясна страна на СЛУ) или целочислени данни (например индекси или глобални идентификатори).

# Работа с вектори

## Пакетите Epetra и EpetraExt

Всеки вектор в Epetra е или последователен или разпределен. Последователните вектори са обикновено малки. Не е изгодно те да бъдат разпределяни между процесорите. Често последователните вектори се повтарят в различните процесори. От друга страна, разпределените вектори са големи и техните елементи се разпределят върху процесорите. За целта потребителят трябва да зададе използваното разделяне. Това става като се използва комуникатор, и обект от Epetra наречен *map* (съответствие). Това съответствие е всъщност разделяне на списък от глобални идентификатори.

# Работа с вектори

## Комуникатори

Виртуалният базов клас `Eperta_Comm` е интерфейс, който съдържа общата информация и услуги, необходими на другите класове от Eperta да работят на последователни или паралелни компютри. Комуникатор е необходим за създаването на всички обекти `Eperta_Map`, които от своя страна трябват за създаването на останалите обекти от Epetra. Интерфейсът `Eperta_Comm` има две реализации:

- `Eperta_SerialComm` за серийни програми и
- `Eperta_MpiComm` за разпределени MPI програми

# Работа с вектори

## Комуникатори

Ето начина за създаване на комуникатор на Eperta.

```
#include "Epetra_ConfigDefs.h"
#include "mpi.h"
#include "Epetra_MpiComm.h"
int main( int argv, char *argv[]) {
  MPI_Init(&argc, &argv);
  Epetra_MpiComm Comm(MPI_COMM_WORLD);
...
  MPI_Finalize();
}
```

# Работа с вектори

## Комуникатори

Повечето методи на комуникатора наподобяват функциите от MPI. Класът предоставя функции като

- `MyPID()` – Идентификатор на процесор,
- `NumProc()` – Общ брой на процесорите,
- `Barrier()` – Бариера (изпълнението на програмата продължава след като всички процесори са извикали тази функция,
- `Broadcast()` – Изпращане до всички,
- `SumAll()` – Сумиране на стойности от всички процесори,
- `GatherAll()` – Получаване от всички процесори,
- `MaxAll()` – намиране на максимум

# Работа с вектори

## Съответствия

Съответствието (map) е разпределение на целочислени елементи върху процесорите. класът Eperta_Map съдържа методи, които ни предоставят:

- Глобален брой на елементите,
- Локален брой на елементите и
- Глобално номериране на локалните елементи.

# Работа с вектори

## Съответствия

Има три начина да се създаде съответствие. Най простият е да се зададе само глобалният брой на елементите:

```
Epetra_Map Map(NumGlobalElements,0,Comm);
```

В този случай конструкторът приема за параметри общия брой елементи, началния индекс (най-често 0 или 1) и комуникатор.
Вторият начин да се задават локалните елементи:

```
Epetra_Map Map(-1,NumMyElements,0,Comm);
```

# Работа с вектори

## Съответствия

Третият най-общ начин за създаването на съответствие е на всеки процесор да се задават локалните елементи, както и глобалната им номерация:

```
#include "Epetra_Map.h"
// ...
MyPID = Comm.MyPID();
switch( MyPID ) {
case 0:
  MyElements = 2;
  MyGlobalElements = new int[MyElements];
  MyGlobalElements[0] = 0;
  MyGlobalElements[1] = 4;
  break;
case 1:
  MyElements = 3;
  MyGlobalElements = new int[MyElements];
  MyGlobalElements[0] = 1;
  MyGlobalElements[1] = 2;
  MyGlobalElements[2] = 3;
  break;
}
Epetra_Map Map(-1,MyElements,MyGlobalElements,0,Comm);
```

# Работа с вектори

## Съответствия

Веднъж конструиран обекта, той може да бъде запитван за броя глобални и локални елементи, както и за съответствието:

```
int NumGlobalElements = Map.NumGlobalElements();
int NumMyElements = Map.NumMyElements();
int * MyGlobalElements = Map.MyGlobalElements();
```

# Работа с вектори

## Създаване на вектори

В Epetra е възможно да се дефинират последователни вектори, дори и за паралелните платформи.
Последователен вектор е такъв, който според програмиста няма нужда да бъде разпределен върху процесорите.
Забележете, че всеки процесор създава свой собствен последователен вектор. Като се променят елементите на един от процесорите, векторите върху другите процесори не се променят.

# Работа с вектори

## Създаване на вектори

Последователни вектори се създават посредством класовете Eperta_SerialDenseVector и Eperta_IntSerialDenseVector:

```
#include "Epetra_IntSerialDenseVector.h"
#include "Epetra_SerialDenseVector.h"
//...
  Epetra_SerialDenseVector DoubleVector(Length);
  Epetra_SerialIntDenseVector IntVector(Length);
//...
```

# Работа с вектори

## Създаване на вектори

Разпределен вектор е обект, чиито елементи са разпределени върху повече от един процесор. Разпределените обекти в Epetra се конструират посредством съответствие (map). Например:

```
Eperata_Vector x(Map);
```

Този конструктор заделя памет за вектора и нулира всички негови елементи.

# Работа с вектори

## Създаване на вектори

Също така може да се ползва и копи-конструктора:

```
Eperata_Vector y(x);
```

Веднъж създаден вектора, неговите елементи могат да се адресират посредством оператора [], независимо от начина на конструиране, например

```
x[i] = 1.0*i;
```

Тук i е локален индекс.

# Работа с вектори

## Операции с вектори

Ето и някои от методите, които могат да се използват с разпределените вектори:

- `int NumMyElements()` – връща локалната дължина на вектора,
- `int NumGlobalElements()` – връща глобалната дължина на вектора,
- `int Norm1(double *Result)` – връща 1-нормата,
- `int Dot(const Eperta_MultiVector &A, double *Result)` – пресмята скаларното произведение на два вектора,
- `int Scale(double ScalarA, const Eperta_MultiVector &A)` – заменя стойностите на вектора със стойностите от вектора A умножение по ScalarA,
- `int PutScalar(double Scalar)` – Присвоява Scalar на всички елементи от вектора.

# Работа с матрици

## Матрици

Eperta съдържа няколко различни класа за матрици. И матриците могат да бъдат както последователни, така и разпределени. Последователна матрица може да е например матрицата, съответстваща на даден елемент в крайно-елементна дискретизация. Тези матрици са с относителни малки размери, така че е безсмислено тяхното разпределяне.

Други матрици, например матрицата на СЛУ, трябва да бъдат разпределени, за да се постигне разпаралеляване. Базовия клас

`Eperta_RowMatrix` е предназначен за достъп по редове до матрици с плаваща точка.

# Работа с матрици

## Видове матрици

Следва списък с класовете, които наследяват този клас:

- `Eperta_CrsMatrix` – за поточкови матрици;
- `Eperta_VbrMatrix` – за блочни матрици;
- `Eperta_FECrsMatrix` и `Eperta_FEVbrMatrix` — за матрици, получени по МКЕ.

# Работа с матрици

## Създаване на матрици

Един от начините за създаване на последователна плътна матрица с размери n на m е:

```
#include "Epetra_SerialDenseMatrix.h"
//...
Epetra_SerialDenseMatrix D(n, m);
```

Разбира се, може да се създаде обект с нулеви размери и размерите да се зададат в последствие:

```
Epetra_SerialDenseMatrix D;
//...
D.Shape(n, m);
D.ReShape(m, n);
```

# Работа с матрици

## Създаване на матрици

Ето пример за използването на плътна последователна матрица:

```
int NumRowsA = 2, NumColsA = 2;
int NumRowsB = 2, NumColsB = 1;
Epetra_SerialDenseMatrix A, B;
A.Shape(NumRowsA, NumColsA);
B.Shape(NumRowsB, NumColsB);
// ... here set the elements of A and B
Epetra_SerialDenseMatrix AtimesB;
AtimesB.Shape(NumRowsA,NumColsB);
double alpha = 1.0, beta = 1.0;
AtimesB.Multiply('N','N',alpha, A, B, beta);
cout << AtimesB;
```

Методът `Multiply()` Извършва операцията $C = aA + bB$, където матриците, се транспонират, ако се сменят параметрите 'N' с 'T'.

# Работа с матрици

## Създаване на матрици

За решението на СЛУ с плътна матрица, трябва да се създаде обект от типа `Epetra_SerialDenseSolver`. Ето как става това:

```
Epetra_SerialDenseSolver Solver();
Solver.SetMatrix(D);
Solver.SetVectors(x, b);
```

След това е възможно да се обърне матрицата с `Invert()`, да се реши линейната система чрез `Solve()`, да се извърши итеративно уточняване на решението с `ApplyRefinement()`.

# Работа с матрици

## Създаване на матрици

В библиотеката Epetra има класове за работа с разпределени разредени матрици. Тези класове позволяват, както добавяне на елементи ред по ред, така и добавяне елемент по елемент. На лице е поддръжка за основните операции, в това число скалиране, норма, умножение на матрица по вектор. Приложенията, използващи обектите от Epetra, не се нуждаят от допълнителна информация за използвания начин за съхранение на данните. Epetra използва два основни формата, един подходящ за поточкови матрици и друг — за блочни. В този раздел ще представим поточковия вариант. Други формати на матрици могат да се дефинират от потребителя, наследявайки `Epetra_RowMatrix`.

# Работа с матрици

## Операции с матрици

Ето и някои от методите на Epetra_RowMatrix, извършващи математически операции:

- `virtual int Multiply (bool TransA, const Epetra MultiVector &X,Epetra MultiVector &Y) const=0`
  Връща резултата от умножението на матрицата с вектор X във вектор Y.
- `virtual int Solve (bool Upper, bool Trans, bool UnitDiagonal, constEpetra MultiVector &X, Epetra MultiVector &Y) const=0`
  Връща резултата от *локално решаване* използвайки триъгълна матрица.
- `virtual int InvRowSums (Epetra Vector &X) const=0`
  Пресмята сумата от абсолютните стойности на редовете на матрицата. Резултатът се връща в X.
- `virtual int LeftScale (const Epetra Vector &X)=0`
  Скалира матрицата с вектора X отляво.
- `virtual int InvColSums (Epetra Vector &X) const=0`
  Пресмята сумата от абсолютните стойности на колоните на матрицата. Резултатът се връща в X.
- `virtual int RightScale (const Epetra Vector &X)=0`
  Скалира матрицата с вектора X отдясно.

# Работа с матрици

## Пример

Започваме със съответствие (Map) и дефинираме локалния брой на редовете, както и глобалната номерация за всеки ред.

```
Epetra_Map Map(NumGlobalElements,0,Comm);
int NumMyElements = Map.NumMyElements();
int *MyGlobalElements = Map.MyGlobalElements( );
```

В частност, имаме, че `j=MyGlobalElements[i]` е глобалният номер на локалния ред `i`. Следващата стъпка е да зададем броя на ненулеви елементи за ред. Това може да стане по два начина — единият е да се зададе едно число за всички (локални) редове, а другият е да се зададе за всеки ред поотделно — чрез масив.

# Работа с матрици

## Пример

При първия вариант матрицата се създава по следния начин:

```
Epetra_CrsMatrix A(Copy,Map,3);
```

Ето и пример за втория начин:

```
int * NumNz = new int[NumMyElements];
for( int i=0 ; i<NumMyElements ; i++ )
  if( MyGlobalElements[i]==0 ||
      MyGlobalElements[i] == NumGlobalElements-1)
    NumNz[i] = 2;
  else
    NumNz[i] = 3;
```

# Работа с матрици

## Пример

Сега командата за създаване на матрица е:

```
Epetra_CrsMatrix A(Copy,Map,NumNz);
```

Добавяме редовете един по един:

```
for( int i=0 ; i<NumMyElements; ++i ) {
  if (MyGlobalElements[i]==0) {
    Indices[0] = 1;
    NumEntries = 1;
  } else if (MyGlobalElements[i] ==
      NumGlobalElements-1) {
    Indices[0] = NumGlobalElements-2;
    NumEntries = 1;
  } else {
    Indices[0] = MyGlobalElements[i]-1;
    Indices[1] = MyGlobalElements[i]+1;
    NumEntries = 2;
  }
```

# Работа с матрици

## Пример

```
    A.InsertGlobalValues(MyGlobalElements[i],
                         NumEntries,
                         Values, Indices);
    // Put in the diagonal entry
    A.InsertGlobalValues(MyGlobalElements[i],
                         1, &two,
                         MyGlobalElements+i);
}
```

Забележете, че индексите на колоните са глобални. Най-накрая трансформираме организацията на матрицата в такава, базирана на локални индекси. Тази трансформация е нужна, за да се изпълнява паралелно и ефективно умножението на матрица по вектор, както и другите матрични операции.

```
A. FillComplete();
```

# Работа с матрици

## Други методи

Ето и още някои полезни методи:

- `virtual bool Filled () const=0` Връща `true` ако `FillComplete()` е била извикана и `false` в противен случай
- `virtual double NormInf () const=0` Връща $C$ нормата на глобалната матрица
- `virtual double NormOne () const=0` Връща едно-нормата на глобалната матрица
- `virtual int NumGlobalNonzeros () const=0` Връща броя на ненулевите елементи в глобалната матрица.
- `virtual int NumGlobalRows () const=0` Връща броя на глобалните редове на матрицата.
- `virtual int NumGlobalCols () const=0` Връща броя на глобалните колони на матрицата
- `virtual int NumGlobalDiagonals () const=0` Връща броя на глобалните ненулеви елементи разположени по главния диагонал (базирайки се на сравнение на броя на глобалните редове и колони)
- `virtual int NumMyNonzeros () const=0` Връща броят на ненулевите елементи на локалния процесор

# Работа с матрици

## Други методи

- `virtual int NumMyRows () const=0` Връща броя на локалните редове от матрицата
- `virtual int NumMyCols () const=0` Връща броя на локалните колони от матрицата
- `virtual int NumMyDiagonals () const=0` Връща броя на локалните ненулеви елементи по главния диагонал;
- `virtual bool LowerTriangular () const=0` Връща `true` ако матрицата е локално долно триъгълна и `false` в противен случай.
- `virtual bool UpperTriangular () const=0` Връща `true` ако матрицата е локално горно триъгълна и `false` в противен случай.
- `virtual const Epetra_Map & RowMatrixRowMap () const=0` Връща съответствие, асоциирано с редовете на матрицата
- `virtual const Epetra_Map & RowMatrixColMap () const=0` Връща съответствие, асоциирано с колоните на матрицата

# AztecOO

## AztecOO

AztecOO е предназначен за решаване на системи линейни уравнения от вида

$$AX = B,$$

където $A \in \mathbb{R}^{n \times n}$ е матрицата на системата, $X$ е решението, а $B$ е дясната част. Въпреки че AztecOO може да бъде използван и без Epetra, тук предполагаме, че $A$ е `Epetra_RowMatrix`, и че и двата вектора $X$ и $B$ са `Epetra_Vector` или `Epetra_MultiVector`.

# AztecOO

## Основна употреба

Първо ще очертаем стъпките за прилагането на AztecOO към система линейни уравнения. Като за начало трябва да бъде създаден обект `Epetra_LinearProblem` с командата

```
Epetra_LinearProblem Problem(&A, &x, &b);
```

Тук $A$ е матрица от Epetra, а $x$ и $b$ са Epetra вектори. След това е необходимо да се създаде обект от класа `AztecOO`.

```
AztecOO Solver(Problem);
```

Сега трябва да се зададе как да бъде решена линейната система. Всички настройки на AztecOO се задават чрез два масива — един от цели числа (int), а другият от числа с плаваща точка (double).

# AztecOO

## Настройки

Ето как се задават настройките по подразбиране.

```
int options[AZ_OPTIONS_SIZE];
double params[AZ_OPTIONS_SIZE];
AZ_defaults(options, params);

Solver.SetAllAztecOptions(options);
Solver.SetAllAztecParams(params);
```

Последните две извикани функции копират стойностите на опциите и параметрите във вътрешни променливи за обекта `Solver`.

# AztecOO

## Настройки

Възможно е и задаване на параметрите и без създаване на *options* и *params*, използвайки функциите `SetAztecOption()` и `SetAztecParams`. Например със следния израз,

```
Solver.SetAztecOption( AZ_precond, AZ_Jacobi );
```

се задава поточков пробусловител на Якоби.

# AztecOO

## Решаване

Накрая следва и самото решаване. За да се реши системата с максимум 2012 итерации и относителна точност $10^{-9}$ се извиква следния метод

```
Solver.Iterate(2012, 1.e-9);
```

След завършването на `Iterate()`, броят на итерациите и крайният резидуал могат да се прочетат съответно с `Solver.NumIters()` и `Solver.TrueResidual()`.

# Заключение

## Приятна пролет с Блугена!

Въпреки голямото разнообразие от библиотеки са необходими значителни усилия за навръзването им за постигане на желаните резултати!