Development of a General Minimal RESidual (GMRES) method alogrithm for Compressed Sparse Row (CSR) Matrices and implementation of ILU0 preconditioning in Fortran90

Alessandro Marcelli
A.A. 2022-2023

Compressed Sparse Row Matrices

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

```
nzval  = [ 5 8 3 6 ]
colind = [ 1 2 3 2 ]
rowpnt = [ 0 1 2 3 4]
```

A Matrix is called Sparse when most of its elements are zero.
Sparse data is more easily compressed and thus operations are much less resource intensive compared to Dense Matrices.

One possible format for storage of a Sparse Matrix is the Compressed Sparse Row one, which employs three 1D vectors:

- nzval: nonzero values of the matrix
- colind: column indices of nonzero values
- rowpnt: total number on nonzero values above row j

Hessemberg Matrices

Special kind of square matrices that are “almost triangular” that either have null elements

- above the first superdiagonal (Lower Hessemberg Matrix)
- below the 1st subdiagonal (Upper Hessemberg Matrix)

If a Hessemberg Matrix has no null elements in said super/subsiagonal, is said to be unreduced.

$$A = \begin{pmatrix} 1 & 4 & 2 & 3 \\ 3 & 4 & 1 & 7 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \end{pmatrix}$$

Unreduced Upper Hessemberg Matrix

$$B = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 5 & 2 & 3 & 0 \\ 3 & 4 & 3 & 7 \\ 5 & 6 & 1 & 1 \end{pmatrix}$$

Unreduced Lower Hessemberg Matrix

$$C = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 5 & 2 & 0 & 0 \\ 3 & 4 & 3 & 7 \\ 5 & 6 & 1 & 1 \end{pmatrix}$$

Lower Hessemberg Matrix

Systems of linear equations and differential equations

A n-th equations linear system is defined as such

$$\begin{cases} \sum_{i=1}^{n} A_{1i} x^i = b^1 \\ \vdots \\ \sum_{i=1}^{n} A_{ni} x^i = b^n \end{cases}$$

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

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \end{pmatrix} = \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^n \end{pmatrix}$$

Partial Differential Equations (PDEs) and in particular Ordinary Differential Equations (ODEs) problems such as Cauchy or Sturm-Liouville ones are examples for systems of equations.

$$\begin{cases} a_2(t)\ddot{x}(t) + a_1(x)\dot{x}(t) + a_0 x(t) = f(t) \\ \dot{x}(t_0) = \dot{x}_0 \\ x(t_0) = x_0 \end{cases}$$

$$\begin{cases} a_2(t)\ddot{x}(t) + a_1(x)\dot{x}(t) + a_0 x(t) = f(t) \\ x(t_0) = x_0 \\ x(t_1) = x_1 \end{cases}$$

ODEs of order greater than one can be rewritten as linear systems of 1st order differential equations.

$$m\ddot{x}(t) + c\dot{x}(t) + k\,x(t) = 0 \;\rightarrow\; \begin{cases} \dot{x}(t) = v(t) \\ \dot{v}(t) = -c\,v(t) - kx(t) \end{cases} \;\rightarrow\; \begin{pmatrix} \dot{x}(t) \\ \dot{v}(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -k & -c \end{pmatrix} \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}$$

General Minimum RESidue overview

Iterative method for the numerical solution of an indefinite nonsymmetric system of linear equations $A\vec{x}=\vec{b}$
The solution is approximated with a vector with minimal residual in a Krylov subspace defined as

$$K_n=K_n(A,\vec{r}_0)=span\{\vec{r}_0, A\vec{r}_0, A^2\vec{r}_0,\dots,A^{n-1}\vec{r}_0\}$$

$\vec{r}_0=\vec{b}-A\vec{x}_0$ Is the initial error given by the Ansatz $\vec{x}_0$

This basis for K_n is at risk of linear dependency.
Using the Arnoldi iteration one can find a better one

$$\{\vec{q}_1,\dots,\vec{q}_n\} \;;\; \vec{q}_1=\frac{\vec{r}_0}{\beta} \;;\; \beta=\|r_0\|_2$$

The Arnoldi iteration yelds two matrices

$$Q_n=\begin{pmatrix} q_1^1 & q_2^1 & \cdots & q_n^1 \\ q_2^1 & \vdots & \vdots & q_n^2 \\ \vdots & \vdots & \vdots & \vdots \\ q_1^m & q_2^m & \cdots & 1_n^m \end{pmatrix}=(\vec{q}_1\vec{q}_2\cdots\vec{q}_n)$$

$\widetilde{H}: AQ_n=Q_{n+1}\widetilde{H}_n$ ← (n+1)-by-n Hessember Matrix

Using the results from Arnoldi we can write the solution as

$$\vec{x}_n=\vec{x}_0+Q_n\vec{y}_n \;\;;\;\; \vec{y}_n\in R^n$$

$$\|\vec{r}_n\|=\|\vec{b}-A\vec{x}_n\|$$

$$\|\vec{r}_n\|=\|\vec{b}-A(\vec{x}_0+Q_n\vec{y}_n)\|$$

$$\|\vec{r}_n\|=\|\vec{r}_0-AQ_n\vec{y}_n\|$$

$$\vec{e}_1=(1 \quad 0 \quad \cdots \quad 0)^T$$

$$\|\vec{r}_n\|=\|\beta Q_{n+1}\vec{e}_1-Q_{n+1}\widetilde{H}_n y_n\|=\|\beta\vec{e}_1-\widetilde{H}_n y_n\|$$

Now we solve the linear least squares problem obtained, which gives us the y_n needed for computing x_n.
If r_n is not small enough, we repeat the process using the last residue as r_0.

Convergence

$K_n \subset K_{n+1}$ → The residual does not increase.

For a m-by-m matrix A, at the m-th iteration the K_m subspace is equal to R^m and thus the method converges to exact solution, but one can often get a “good enough” approximation before that.

Greenbaum, Pták and Strakoš theorem states the possibility of the method stalling for m-1 steps and suddendly converging to zero at the last step.
This is not a common occurrence though, and the method usually works well, expecially when the eigenvalues of A are clustered away from the origin and A is not too far from normality.

GMRES recap and computational costs

STEPS RECAP

1) Calculation of q_n with Arnoldi
2) Minimization of r_n
3) Computation of solution
4) Repeat if r_n too big
5) Output generation

COMPUTATIONAL COSTS

1) $2m^2$ floating point operations for dense matrices, O(m) for CSR ones
2) O(nm) floating point operations at the n-th iteration

Least squares problem solution (1)

$$\|\vec{r}_n\| = \|\beta \vec{e}_1 - \widetilde{H}_n y_n\| = \|\widetilde{H}_n y_n - \beta \vec{e}_1\|$$

(n+1)-by-n dimension matrix, gives over-constrained linear system of n+1 equation for n variables

We compute the minimum using a QR decomposition

$$\Omega_n \widetilde{H}_n = \widetilde{R}_n$$ ← (n+1)-by-n upper triangular matrix

(n+1)-by-(n+1) orthogonal matrix

The upper triangular matrix can be rewritten as

$$\widetilde{R}_n = \begin{bmatrix} R_n \\ 0 \end{bmatrix}$$ ← n-by-n upper triangular matrix

The decomposition is easily updatable by virtue of the following property of Hessenberg matrices

$$\widetilde{H}_{n+1} = \begin{bmatrix} \widetilde{H}_n & h_{n+1} \\ 0 & h_{n+2,n+1} \end{bmatrix}$$ ← $h_{n+1} = \begin{pmatrix} h_{1,n+1} & \cdots & h_{n+1,n+1} \end{pmatrix}^T$

$$\begin{bmatrix} \Omega_n & 0 \\ 0 & 1 \end{bmatrix} \widetilde{H}_{n+1} = \begin{bmatrix} R_n & r_{n+1} \\ 0 & \rho \\ 0 & \sigma \end{bmatrix}$$

If we manage to set this to zero, we get a triangular matrix!

We need the Givens Rotation for this

$$G_n = \begin{bmatrix} I_n & 0 & 0 \\ 0 & c_n & s_n \\ 0 & -s_n & c_n \end{bmatrix} \; ; \; c_n = \frac{\rho}{\sqrt{\rho^2 + \sigma^2}} \; , \; s_n = \frac{\sigma}{\sqrt{\rho^2 + \sigma^2}}$$

Least squares problem solution (2)

Using the Givens rotation we defined we can write

$$\Omega_{n+1} = G_n \begin{bmatrix} \Omega_n & 0 \\ 0 & 1 \end{bmatrix}$$

This is now a triangular matrix!

$$\Omega_{n+1}\widetilde{H}_{n+1} = \begin{bmatrix} R_n & r_{n+1} \\ 0 & r_{n+1,n+1} \\ 0 & 0 \end{bmatrix} \quad ; \quad r_{n+1,n+1} = \sqrt{\rho^2+\sigma^2}$$

$$\vec{g}_n \in R^n$$
$$\vec{\gamma}_n \in R$$

The minimization problem thus becomes

$$\|\vec{r}_n\| = \|\widetilde{H}_n y_n - \beta \vec{e}_1\| = \|\Omega_n(\widetilde{H}_n y_n - \beta \vec{e}_1)\| = \|\widetilde{R}_n y_n - \beta\Omega_n \vec{e}_1\| = \|\widetilde{R}_n y_n - \vec{\tilde{g}}_n\| \quad ; \quad \vec{\tilde{g}}_n = \begin{bmatrix} \vec{g}_n \\ \vec{\gamma}_n \end{bmatrix}$$

And the vector that minimizes the problem is given by

$$y_n = R_n^{-1} g_n$$

Preconditioning

Preconditioning is a procedure that helps iterative methods to converge faster.
It consists in applying a matrix, called preconditioner, to the original one, thus obtaining a matrix with a smaller condition number, which means it's less sensible to error during solving.

$$A\vec{x}=\vec{b} \quad \longrightarrow \quad M^{-1}A\vec{x}=M^{-1}\vec{b}$$

We don't need to form this explicitly

$$\hat{d}_0=\hat{r}_0=\hat{b}-UAU^T\hat{x}_0$$

do i=0, n-1

$$\alpha=\frac{\hat{r}_i^T\hat{r}_i}{\hat{d}_i^TUAU^T\hat{d}_i}$$

$$\hat{x}_{i+1}=\hat{x}_i+\alpha\hat{d}_i$$

$$\hat{r}_{i+1}=\hat{r}_i-\alpha UAU^T\hat{d}_i$$

$$\beta=\frac{\hat{r}_{i+1}^T\hat{r}_{i+1}}{\hat{r}_i^T\hat{r}_i}$$

$$\hat{d}_{i-1}=\hat{r}_{i-1}+\beta\hat{d}_i$$

end do

$$\hat{x}=U^{-T}x$$

$$d_i=U^T\hat{d}_i$$

$$r_i=U^{-1}\hat{r}_i$$

$$r_0=b-Ax_0; \qquad d_0=M^{-1}r_0$$

do i=0, n-1

$$\alpha=\frac{r_i^TM^{-1}r_i}{d_i^TAd_i}$$

$$x_{i+1}=x_i+\alpha d_i$$

$$r_{i+1}=r_i-\alpha Ad_i$$

$$\beta=\frac{r_{i+1}^TM^{-1}r_{i+1}}{r_i^TM^{-1}r_i}$$

$$d_{i+1}=M^{-1}r_{i+1}+\beta d_i$$

end do

ILU Preconditioning

One kind of preconditioning the LU factorization (LU)

$A = LU$

Upper Triangular matrix

$U \in R^{nxn}$

$U_{ij} = 0 \quad \forall (i,j) \notin S$

$L\vec{y} = \vec{b} \;\; ; \;\; U\vec{x} = \vec{y}$

Lower Unitriangular matrix

$L \in R^{nxn}$

$L_{ij} = 0 \quad \forall (i,j) \notin S$

Sparsity Pattern

$S \subset \{1, \cdots, n^2\} \;\; , \;\; \{(i.i) : 1 \leqslant i \leqslant n\} \subset S \;\; , \;\; G(A) \subset S$

$G(A) = \{(i.i) \in N^2 : A_{ij} \neq 0\}$

Graph of A

For sparse matrices the LU factorization is not ideal, since the LU factors can be much less sparse than the originl matrix, thus requiring more memory. Instead it's better to implement the Incomplete LU factorization (ILU) since being an approximation it's less memory intensive at the cost of a non-exact result.
This is not a problem, since we're just using it as a preconditioner for an iterative method.

$A \simeq LU$

$M\vec{y} = \vec{b}$

$M = LU$

Fortran90 GMRES code structure

```
subroutine gmres(A_in, b_in, x_out, Iter_in, threshold_in, ip, error)
    *VARIABLE DEFINITION, ALLOCATION AND INITIALIZATION*
    *FIRST RESIDUAL CALCULATION*
    *CHOOSE IF USE PRECONDITION OR NOT*
    do while (error > threshold_in)
     *ARNOLDI ITERATION FOR Iter_in STEPS*
     *MINIMUM SQUARES PROBLEM SOLUTION*
     *OUTPUT CALCULATION*
     *RESIDUAL CROSS CHECK*
    end do
    *DEALLOCATIONS*
end subroutine
```

Slide 12

Slides 13 to 16

Slide 17

Checks if residual is small enough at the end of GMRES, if not the loop starts again

GMRES

GMRES code can be found at
https://github.com/comp-phys-uniroma2/cg/blob/master/gmres.f90

ILU0 Preconditioning

```
select case(ip)
  case('I')
    call getULD_ilu0(A_in, U, L, D)
  case('N')
end select
```

```
subroutine getULD_ilu0(A, U, L, D)
  type(rCSR), intent(in) :: A
  type(rCSR), intent(inout) :: U
  type(rCSR), intent(inout) :: L
  real(dp), intent(inout) :: D(:)

   type(rMSR) :: LU
   integer :: i, j, N, kl, ku, colindj

  print*,'create LU'
  call create(LU, A%nrow, A%nnz+1)

  print*,'compute ilu0'
  call ilu0(A, LU)

  print*,'compute D'
  N = A%nrow
  do i = 1, N
    D(i) = 1.0_dp/LU%nzval(i)
  end do

  print*,'create L,U'
  call create(U, N, (A%nnz-N)/2 + N)
  call create(L, N, (A%nnz-N)/2 + N)

  print*,'compute L,U'
  U%rowpnt(1) = 1
  L%rowpnt(1) = 1
  kl = 0; ku = 0;
  do i = 1, N
    do j = LU%colind(i), LU%colind(i+1) - 1
       colindj = LU%colind(j)
       if (colindj > i) then
         ku = ku + 1
         U%nzval(ku) = LU%nzval(j)
         U%colind(ku) = colindj
       end if
       if (colindj < i) then
         kl = kl + 1
         L%nzval(kl) = LU%nzval(j)
         L%colind(kl) = colindj
       end if
    end do
    kl = kl + 1
    L%nzval(kl) = D(i)
    L%colind(kl) = i
    L%rowpnt(i+1) = kl + 1
    ku = ku + 1
    U%nzval(ku) = D(i)
    U%colind(ku) = i
    U%rowpnt(i+1) = ku + 1
  end do

end subroutine getULD_ilu0
```

Arnoldi Iteration in Fortran90

Iterative eigenvalue algorithm. Starting from an arbitrary vector of norm 1, it finds an approximation of the eigenvalues and eigenvectors for a generic matrix A by constructing an orthonormal basis in the Krylov subspace using the Modified Gram-Schmidt process. Reported here is the k-th step of the iteration

```
mQ(:,1) = r/r_norm
beta(1) = r_norm

do k = 1, Iter_in-1

  ! operates on mH and mQ

  select case(ip)
  case('I')
    call arnoldip(A_in, k, L, U, D)
  case('N')
    call arnoldi(A_in, k)
  end select

  ! eliminate the last element in H ith row and update the rotation matrix
  call apply_givens_rotation(cs, sn, k)

  ! Residual vector update
  beta(k+1) = -sn(k)*beta(k)
  beta(k)   =  cs(k)*beta(k)
  error     = abs(beta(k+1))/b_norm

  print*,k, 'error:', error
  if (error <= threshold_in) then
    exit
  end if

end do
```

1st vector for Arnoldi

K-th step of Arnoldi iteration

ILU0 preconditioning case

No preconditioning case

Givens rotation for QR decomposition. For resource optimization, the matrix mH serves both the role of both H and R matrices, since it's not necessary to keep the old values of either on following iterations.

K-th step of Arnoldi Iteration

```
subroutine arnoldi(A_in, k_in)

    type(rCSR), intent(in)                  :: A_in
    integer, intent(in)                     :: k_in

    real(dp), dimension(:), allocatable     :: q
    real(dp)                                :: q_norm
    integer                                 :: i
    integer                                 :: Qsize

    Qsize = size(mQ, 1)
    allocate(q(Qsize))

    ! new Krylov vector
    call matvec(A_in, mQ(:,k_in), q)

    ! Modified Gram-Schmidt, keeping the Hessenberg matrix
    do i = 1, k_in
      mH(i, k_in) = dot_product(q(:), mQ(:,i))
      q(:)    = q(:) - mH(i, k_in)*mQ(:,i)
    end do

    q_norm = norm2(q)
    mH(k_in+1, k_in) = q_norm
    mQ(:, k_in+1) = q(:) / q_norm

    deallocate(q)

end subroutine arnoldi
```

Modified k-th step Arnoldi with ILU0

```
subroutine arnoldip(A_in, k_in, L, U, D)

  type(rCSR), intent(in)                  :: A_in, L, U
  integer, intent(in)                     :: k_in
  real(dp), intent(in)                    :: D(:)

  real(dp), dimension(:), allocatable     :: q, z
  real(dp)                                :: q_norm
  integer                                 :: i
  integer                                 :: Qsize

  Qsize = size(mQ, 1)
  allocate(q(Qsize))
  allocate(z(Qsize))

  call solveLDU(L,D,U,mQ(:,k_in),z)

  ! new Krylov vector
  call matvec(A_in, z, q)

  deallocate(z)
  ! Modified Gram-Schmidt, keeping the Hessenberg matrix
  do i = 1, k_in
    mH(i, k_in) = dot_product(q(:), mQ(:,i))
    q(:)    = q(:) - mH(i, k_in)*mQ(:,i)
  end do

  q_norm = norm2(q)
  mH(k_in+1, k_in) = q_norm
  mQ(:, k_in+1) = q(:) / q_norm

  deallocate(q)

end subroutine arnoldip
```

ILU factorization every k-th step of the Arnoldi iteration

```
subroutine solveLDU(L,D,U,b,x)
  type(rCSR), intent(in) :: L, U
  real(dp), intent(in) :: D(:), b(:)
  real(dp), intent(inout) :: x(:)

  real(dp), allocatable :: s(:)
  integer :: i

  allocate(s(size(x)))

  call solveL(L,b,s)
  do i = 1, size(x)
    s(i) = D(i)*s(i)
  end do
  call solveU(U,s,x)

  deallocate(s)

end subroutine solveLDU
```

Subroutine for solving a lower triangular linear system by substitution

Subroutine for solving a upper triangular linear system by back-substitution

Givens Rotation

```
subroutine givens_rotation(v1,v2,cs_out,sn_out)

   real(dp), intent(in)  :: v1,v2
   real(dp), intent(out) :: cs_out,  sn_out

   real(dp)              :: t
   t=sqrt(v1**2 + v2**2)

   cs_out = v1/t
   sn_out = v2/t

 end subroutine

 ! ------------------------------------------------------------------
 subroutine apply_givens_rotation(cs, sn, k)

   real(dp), dimension(:), intent(inout)  :: cs, sn
   integer, intent(in)                    :: k

   real(dp)               :: temp
   integer                :: i

   !applied to i-th column
   do i = 1, k-1
     temp   =  cs(i)*mH(i,k) + sn(i)*mH(i+1,k)
     mH(i+1,k) = -sn(i)*mH(i,k) + cs(i)*mH(i+1,k)
     mH(i,k)   = temp
   end do

   ! update the next sin cos values for rotation
   call givens_rotation(mH(k,k), mH(k+1,k), cs(k), sn(k))

   ! eliminate H(i,i-1)
   mH(k,k)   = cs(k)*mH(k,k) + sn(k)*mH(k+1,k)
   mH(k+1,k) = 0.0_dp

 end subroutine
```

$$t = \sqrt{\rho^2 + \sigma^2}$$

$$c_n = \frac{\rho}{\sqrt{\rho^2 + \sigma^2}}$$

$$s_n = \frac{\sigma}{\sqrt{\rho^2 + \sigma^2}}$$

$$\Omega_{n+1} \widetilde{H}_{n+1} = \begin{bmatrix} R_n & r_{n+1} \\ 0 & r_{n+1,n+1} \\ 0 & 0 \end{bmatrix}$$

Residual minimization, output and residual cross-check

```
call solveU(mH(1:k,1:k), beta(1:k), y_gmres)

select case(ip)
  case('I')
    do i = 1, size(x_out)
      tmp_v(i)= dot_product(mQ(i,1:k), y_gmres(1:k))
    end do
    call solveLDU(L,D,U,tmp_v,r)
    x_out = x_out + r
  case('N')
    do i = 1, size(x_out)
      x_out(i)= x_out(i) + dot_product(mQ(i,1:k), y_gmres(1:k))
    end do
end select

! cross check the residual:
call matvec(A_in,x_out,tmp_v)
r = b_in - tmp_v
r_norm = norm2(r)
error = r_norm / b_norm
```

$$y_n = R_n^{-1} g_n$$

ILU0 preconditioning case

$$\vec{x}_n = \vec{x}_0 + Q_n \vec{y}_n \quad ; \quad \vec{y}_n \in R^n$$

No preconditioning case

The loop keeps on going while `error > threshold_in`

Solving the Poisson equation with and without ILU0 1

```
…/cg  master !  17:21  ./poisson 100 1e-3 G 100
Init Matrices
Number of nodes:      1000000
Non zeros:      6940000      6940000
Capacitor lenght:            10
Solve using GMRES
Mem:    1600000000  bytes
=======================
starting GMRES solver
=======================

residual error:    38.296186712274242
residual error:    1.9017370619300762
residual error:   0.10458991769726865
residual error:    5.9180974051615161E-003
residual error:    9.9287269915560592E-004
done!

…/cg  master !  50s  17:22  ./poisson 100 1e-3 G 100 I
Init Matrices
Number of nodes:      1000000
Non zeros:      6940000      6940000
Capacitor lenght:            10
Solve using GMRES
Mem:    1600000000  bytes
=======================
starting GMRES solver
=======================

residual error:    7.1133672187758167E-003
residual error:    8.4291691982361873E-004
done!

…/cg  master !  16s  17:22
```

3d gridle NxNxN

tolerance

Size of GMRES subspace (optional)

Solver:
C= Conjugate Gradient
G= GMRES

Precond: (optional)
J=Jacobi
S=SSOR
I=ILU0

Program for solving a 3D Poisson equation, a famous example of Elliptical PDE using CSR sparse matrices.

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)\varphi(x,y,z)=f(x,y,z)$$

```
call gmres(A_csr, rhs, phi, maxiter, tol, PC, error)
```

Three runs presented:

- 100x100x100
- 150x150x150
- 200x200x200
- … after that my machine crashed, unfortunately

The method converges sensibly faster (36 seconds less of runtime for N=100) when implementing ILU0

Solving the Poisson equation with and without ILU0 2

```
…/cg  master !  23s  21:05  ./poisson 150 1e-3 G 100
Init Matrices
Number of nodes:     3375000
Non zeros:    23490000    23490000
Capacitor lenght:           15
Solve using GMRES
Mem:   1105032704  bytes
======================
starting GMRES solver
======================

residual error:   76.609412339210195
residual error:   14.472142765011723
residual error:   3.0218181102848085
residual error:  0.66419654779097592
residual error:  0.15429856500849065
residual error:   3.6174384024472366E-002
residual error:   8.5507455018786222E-003
residual error:   2.0241811719140259E-003
residual error:   9.9909068451820341E-004
done!
```

```
…/cg  master !  5m26s  21:10  ./poisson 150 1e-3 G 100 I
Init Matrices
Number of nodes:     3375000
Non zeros:    23490000    23490000
Capacitor lenght:           15
Solve using GMRES
Mem:   1105032704  bytes
======================
starting GMRES solver
======================

residual error:   1.7680348939908570
residual error:   1.9510481365808930E-003
residual error:   9.6055046619059905E-004
done!
```

```
…/cg  master !  1m43s  21:28
```

For bigger and bigger Matrices, the vantage become more and more obvious, as expected from theory!

```
…/cg  master !  106ms  18:02  ./poisson 200 1e-3 G 100
Init Matrices
Number of nodes:     8000000
Non zeros:    55760000    55760000
Capacitor lenght:           20
Solve using GMRES
Mem:    -84901888  bytes
======================
starting GMRES solver
======================

residual error:   113.56477290925663
residual error:   30.218785022065333
residual error:   11.181708712910050
residual error:   4.3878822798354076
residual error:   1.7334055898960865
residual error:  0.68711630795579981
residual error:  0.27326931183989994
residual error:  0.10891102793453197
residual error:   4.3491127089832896E-002
residual error:   1.7389538686520819E-002
residual error:   6.9611948602999445E-003
residual error:   2.7887793298325496E-003
residual error:   1.1180343011478011E-003
residual error:   9.9370578455900424E-004
done!
```

```
…/cg  master !  21m18s  18:23  ./poisson 200 1e-3 G 100 I
Init Matrices
Number of nodes:     8000000
Non zeros:    55760000    55760000
Capacitor lenght:           20
Solve using GMRES
Mem:    -84901888  bytes
======================
starting GMRES solver
======================

residual error:   9.3908978020382943
residual error:   4.4406208754141391E-002
residual error:   9.9301237864517482E-004
done!
```

```
…/cg  master !  5m14s  18:29
```