INF5620 Prosjektoppgave:
Løsning av Gross-Pitaevskii-ligningen i en dimensjon

Joachim Berdal Haga

15. mai 2005

1 Fysisk modell

Bose-Einstein-kondensasjon (BEC) beskriver en fase av bosonisk materie der så godt som alle partiklene er i samme kvantetilstand. BEC i tynne gasser av alkaliatomer fanget i en magnetfelle er en typisk eksperimentell realisasjon av fenomenet.

Så lenge temperaturen er godt under den kritiske og tettheten er lav, er de aller fleste atomene i kondensatet og gassen beskrives godt av Gross-Pitaevskii-ligningen. Den er en Schrödingerligning hvor energien har et bidrag for kinetisk energi, et harmonisk oscillator-potensial fra magnetfeltet, og et ytterligere potensial som er proporsjonalt med tettheten til gassen.

2 Matematisk modell

Den tidsuavhengige Gross-Pitaevskii-ligningen ser slik ut:

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V + \frac{4\pi\hbar^2 a}{m}|\Psi|^2\right]\Psi = \mu\Psi$$

hvor μ er energi per partikkel og bølgefunksjonen

$$\Psi = \Psi(\mathbf{r})$$

er normert til antall partikler. Potensialet V, skapt av magnetfellen, er

$$V = V(\mathbf{r}) = \frac{m}{2}(\omega_\perp^2 x^2 + \omega_\perp^2 y^2 + \omega_z^2 z^2).$$

Parameteret a er spredningslengden (dvs, i en forstand, størrelsen) til atomene. Det bestemmer vekselvirkningen i gassen. For $^{87}\mathrm{Rb}$-atomer, et vanlig valg, er spredningslengden i størrelsesorden 100 Bohr-radier.

Skalering til dimensjonsløs form gir

$$\left[-\nabla^2 + V + \gamma|\Psi|^2\right]\Psi = \lambda\Psi$$

$$\Psi = \Psi(\tilde{\mathbf{r}})$$

$$V(\tilde{\mathbf{r}}) = \tilde{x}^2 + \tilde{y}^2 + \omega^2 \tilde{z}^2$$
$$\tilde{\mathbf{r}} = \frac{\mathbf{r}}{\sqrt{\hbar/m\omega_\perp}}, \quad \gamma = \frac{8\pi a}{\sqrt{\hbar/m\omega_\perp}}, \quad \lambda = \frac{2\mu}{\hbar\omega_\perp}, \quad \omega = \frac{\omega_z}{\omega_\perp}$$

I praktiske utregninger velges $a/\sqrt{\hbar\omega_\perp}$ ofte til $4.33 \cdot 10^{-3}$. Men i denne oppgaven vil jeg utforske numeriske egenskaper ved den ikke-lineære ligningen, og da kan det være interessant å sette den betydelig høyere. Fysisk er det verd å poengtere at Gross-Pitaevskii-ligningen kun er gyldig i tynne gasser, der vekselvirkningen bare gir et lite bidrag.

I en dimensjon reduseres det til den ganske enkle (men ikke-lineære) ligningen

$$\left[-\frac{d^2}{dx^2} + x^2 + \gamma|u|^2\right] u = \lambda u$$

Formelt gir det dimensjon $[\text{lengde}]^{-1}$ for spredningslengden a, og det er uklart hvor mye fysisk mening denne varianten av ligningen gir. Men like fullt er det den jeg skal undersøke her.

I fravær av vekselvirkning har vi analytisk løsning på problemet, som da blir en vanlig harmonisk oscillator. Grunntilstanden er

$$u = \left(\frac{n^2}{\pi}\right)^{-1/4} \exp\left\{-\frac{x^2}{2}\right\}, \quad \lambda = 1$$

for n partikler.

3 Numerisk metode

3.1 Endelige elementer

Endelige elementers metode bygger på en approksimasjon ved ortogonale lokale basisfunksjoner,

$$u(x) \approx \hat{u}(x) = \sum_j u_j N_j(x)$$

Hvis vi setter inn approksimasjonen i differensialligningen og kvitter oss med den andrederiverte ved partiell derivasjon, får vi

$$\sum_i \left(\int N_i' N_j' \, dx + \int (x^2 + \gamma \hat{u}^2) N_i N_j \, dx \right) u_j = \lambda \sum_i \left(\int N_i N_j \, dx \right) u_j$$

eller, i matriseformulering,

$$\mathbf{K}(\mathbf{u})\mathbf{u} = \lambda \mathbf{M}\mathbf{u}$$

Elementvis formulering: jeg henviser til oppgave 2.7 fra tidligere, men kort fortalt er elementet

$$\tilde{K}_{i,j}^{(e)} = \int_{-1}^{1} \left(\tilde{N}_i^{(e)\prime} \tilde{N}_j^{(e)\prime} + [q^2(e,\xi) + \hat{u}^2(q(e,\xi))] \tilde{N}_i^{(e)} \tilde{N}_j^{(e)} \right) \det J \, d\xi$$

$$\tilde{M}_{i,j}^{(e)} = \int_{-1}^{1} \tilde{N}_i^{(e)} \tilde{N}_j^{(e)} \mathrm{det} J \; d\xi$$

Høyresiden, $\mathbf{M}$, er konstant og er den vanlige massematrisen. Derimot avhenger venstresiden, $\mathbf{K}(\mathbf{u})$, av den ukjente. Så vi må tilnærme den på et vis.

3.2 Iterativ løsning med hjelp av arpack

Suksessive substitusjoner er kanskje ikke godt språk, men det er ihvertfall metoden.

Arpack løser lineære egenverdiproblemer på en effektiv måte. Så vi gjør om det ikkelineære problemet til en sekvens av lineære egenverdiproblemer som forhåpentligvis konvergerer mot den sanne løsningen.

$$\mathbf{K}(\mathbf{u}^{k-1})\mathbf{u}^k = \lambda \mathbf{M}\mathbf{u}^k$$

Vi trenger ikke noen initiell gjetning for $\mathbf{u}$ med denne metoden. Hvis $\mathbf{u}^0 = 0$, vil første iterasjon løse det lineære problemet $(-d^2/dx^2 + x^2)u = \lambda u$, så vi kan spare en iterasjon ved å bruke den analytiske løsningen på det problemet for å konstruere $\mathbf{K}(\mathbf{u}^0)$.

3.3 Korrektiv løsning med Newton-Raphson

Jeg formulerer her Newtons metode for egenverdiproblemet på matrisenivå. Det er ikke åpenbart for meg hvordan den formuleres på PDE- eller FEM-nivå for egenverdiproblemer.

Generelt går Newtons metode ut på å beregne en korreksjonsvektor til den foreløpige løsningen ved å følge gradienten til dens nullpunkt.

$$\delta\mathbf{u} = \mathbf{J}^{-1}\mathbf{F}$$

I et ordinært (ikke-egenverdi) problem vil $\mathbf{F}$ være roten til ligningen

$$\mathbf{F_1}(\mathbf{u}; \lambda) = (\mathbf{K} - \lambda\mathbf{M})\mathbf{u} = 0$$

(undertrykker her og heretter $\mathbf{u}$-avhengigheten til $\mathbf{K}$).

Men for å få med egenverdien må ligningssystemet utvides med en ligning. Ligningen gir normen til løsningen, $\mathbf{u}^T\mathbf{u} = c$. Normen er entydig bestemt av antall partikler i kondensatet og antall noder per lengdeenhet.

$$\mathbf{F} = \begin{bmatrix} \mathbf{F_1} \\ F_2 \end{bmatrix} = \begin{bmatrix} (\mathbf{M} - \lambda\mathbf{K})\mathbf{u} \\ \mathbf{u}^T\mathbf{u} - c \end{bmatrix}$$

Hvis $\mathbf{J}$ settes opp elementvis, ser man at de kan sammenfattes slik:

$$J_{i,j} = \begin{cases} \partial f_i/\partial u_j = K_{i,j} - \lambda M_{i,j} & i,j = 1 \ldots n \\ \partial f_i/\partial u_j = 2u_j & i = n+1,\ j = 1 \ldots n \\ \partial f_i/\partial \lambda = -\sum_k M_{i,k} u_k & i = 1 \ldots n,\ j = n+1 \\ 0 & i = j = n+1 \end{cases}$$

hvor n er antall ukjente i $\mathbf{u}$.

For å finne korreksjonsvektoren til $\mathbf{F}$ må man altså løse ligningssystemet

$$\begin{bmatrix} \mathbf{K} - \lambda\mathbf{M} & -\mathbf{M}\mathbf{u} \\ 2\mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \delta\mathbf{u} \\ \delta\lambda \end{bmatrix} = - \begin{bmatrix} (\mathbf{K} - \lambda\mathbf{M})\mathbf{u} \\ \mathbf{u}^T\mathbf{u} - c \end{bmatrix}$$

Merk at matrisen på venstre side ikke er symmetrisk og ikke har båndstruktur. En effektiv løsningsmetode må ta hensyn til det.

Etter litt regning ser vi at dette gir

$$\mathbf{K}(\mathbf{u} + \delta\mathbf{u}) - (\lambda + \delta\lambda)\mathbf{M}(\mathbf{u} + \delta\mathbf{u}) = -\delta\lambda\mathbf{M}\delta\mathbf{u}$$

dvs. $(\mathbf{u} + \delta\mathbf{u}, \lambda + \delta\lambda)$ oppfyller egenverdiligningen modulo en andreordens korreksjon, og

$$(\mathbf{u} + \delta\mathbf{u})^T(\mathbf{u} + \delta\mathbf{u}) - c = \delta\mathbf{u}^T\delta\mathbf{u}$$

viser at $\mathbf{u} + \delta\mathbf{u}$ oppfyller normbetingelsen, også med en andreordens korreksjon.

Det vil si at hvis vi starter med en gjetning som er tilstrekkelig nær løsningen, og itererer over ligningssystemet over, bør vi få kvadratisk konvergens — ihvertfall i det lineære tilfellet.

Med denne metoden er det viktig at $\mathbf{u}^0$ er nær løsningen, hvis ikke konvergerer den ikke. Eller den konvergerer mot feil løsning — en annen tilstand enn grunntilstanden. Det er naturlig også her å bruke den analytiske løsningen som $\mathbf{u}^0$.

4 Analyse

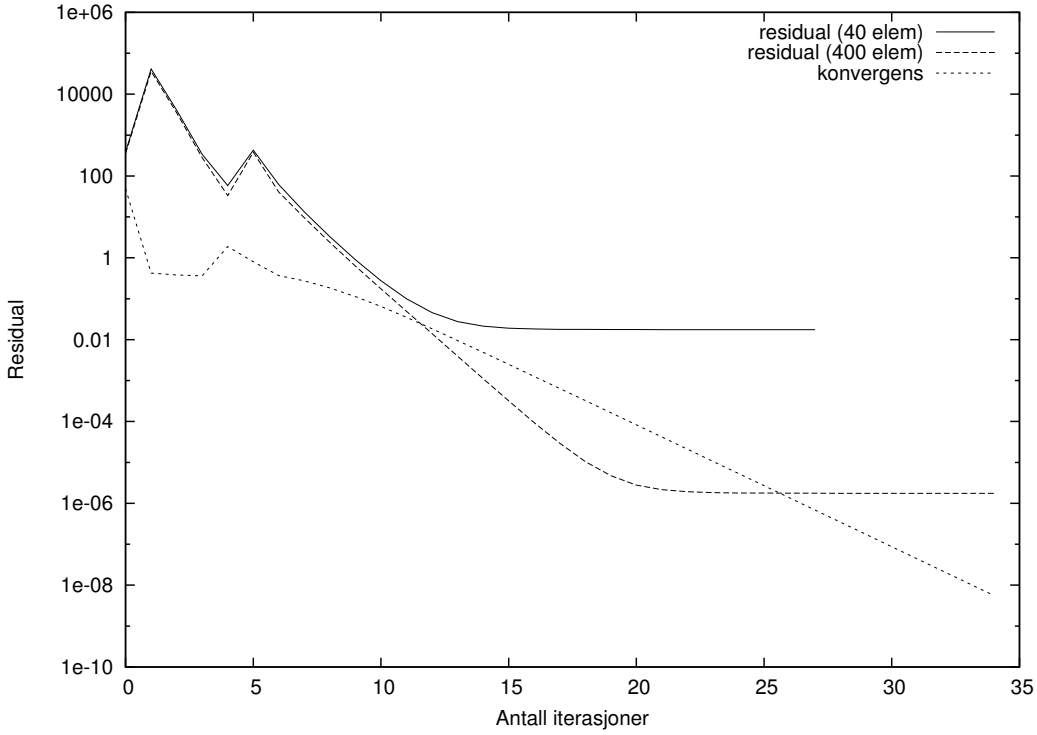
4.1 Konvergens

Så lenge problemet konvergerer, kan vi gjøre konvergensfeilen vilkårlig liten ved å tilstrekkelig mange iterasjoner. Konvergensfeilen er her målt ved endringen fra iterasjon til iterasjon, og er i praksis et grovt mål på avstanden mellom løsningen av det lineariserte problemet og løsningen på det ikkelineære problemet.

Men vi har også en diskretiseringsfeil som avhenger av Δx i diskretiseringen. Figur 1 viser konvergens (lineariseringsfeil) og residual for to forskjellige gittere. Vi ser at residualet når et platå og slutter å synke. Dette platået er diskretiseringsfeilen.

I figur 2 sammenlignes de to metodene jeg har brukt for å løse problemet. Der vises det at Newtonmetoden bruker mange flere iterasjoner på å nå samme presisjon. Sammenligningen er litt urettferdig: Den iterative metoden kaller arpack, som internt gjør et antall iterasjoner hver gang. Likevel er sannheten ikke langt unna. Den iterative metoden er for øyeblikket raskere, spesielt på større problemer.

Andre problemer med den iterative metoden er at den lett konvergerer mot andre tilstander enn grunntilstanden, og at den ikke konvergerer for sterke ikkelineariteter. Jeg ser for meg at begge disse problemene lar seg løse ved å innføre et kontinuasjonsparameter på vekselvirkningsstyrken γ, siden vi vet den eksakte løsningen når den forsvinner, og siden endringen i løsningen er gradvis med endring i denne (se figur 4 og 5). Jeg har ikke eksperimentert med det på grunn av tidsnød (implementasjonen er enkel, men testing tar tid).



Figur 1: *Konvergens og residual*

Figur 3 viser at residualet går omtrent som Δx^4, som er overraskende. Forklaringen kan ligge i at utregningen av residualet gjøres ved endelig differanse-metoden, som selv er av andre orden i Δx.

Dette kan også forklare hvorfor residualet i figur 1 tilsynelatende faller raskere enn lineariseringsfeilen.

4.2 Fysiske egenskaper

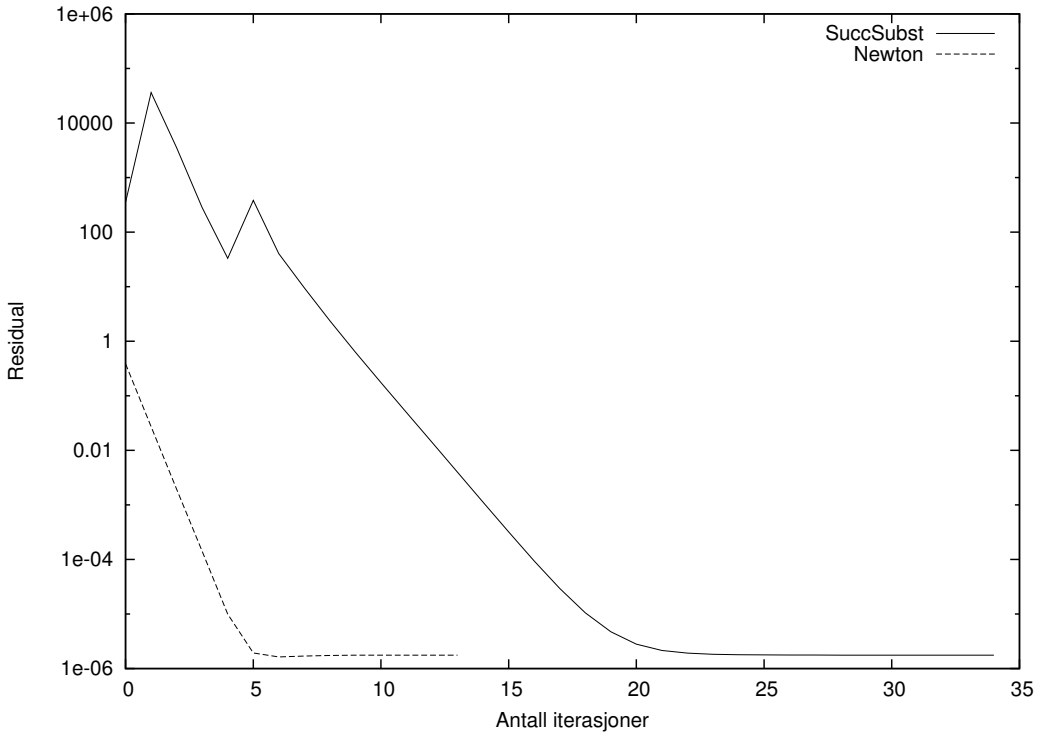
Som jeg sa tidligere har jeg ikke lyst til å forsøke å trekke for mye fysikk ut av løsningen av det endimensjonale problemet. Men jeg liker fine bilder.

Figur 4 viser tettheten i kondensatet. Ved $\gamma = 0$ har vi en ordinær Gausskurve. Ettersom vekselvirkningsstyrken øker, ser vi at toppen blir lavere og massen spres mer utover. Det er konsistent med at partiklene frastøter hverandre mer; de forsøker å maksimere avstanden til naboene.

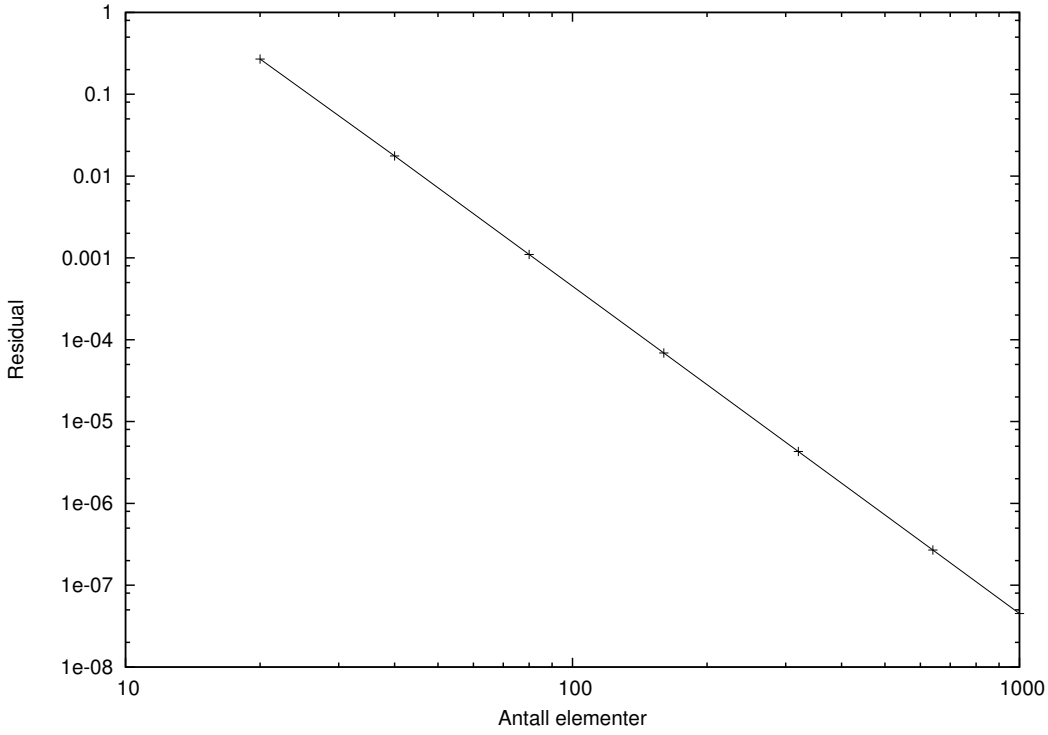
$\gamma = 0.0109$ er den fysisk relevante vekselvirkningsstyrken. Vi ser at korreksjonen i det tilfellet er forholdsvis liten.

Figur 5 viser egenverdiene for forskjellige antall partikler, med moderat vekselvirkningsstyrke. Endringen er gradvis ved økende partikkeltall.

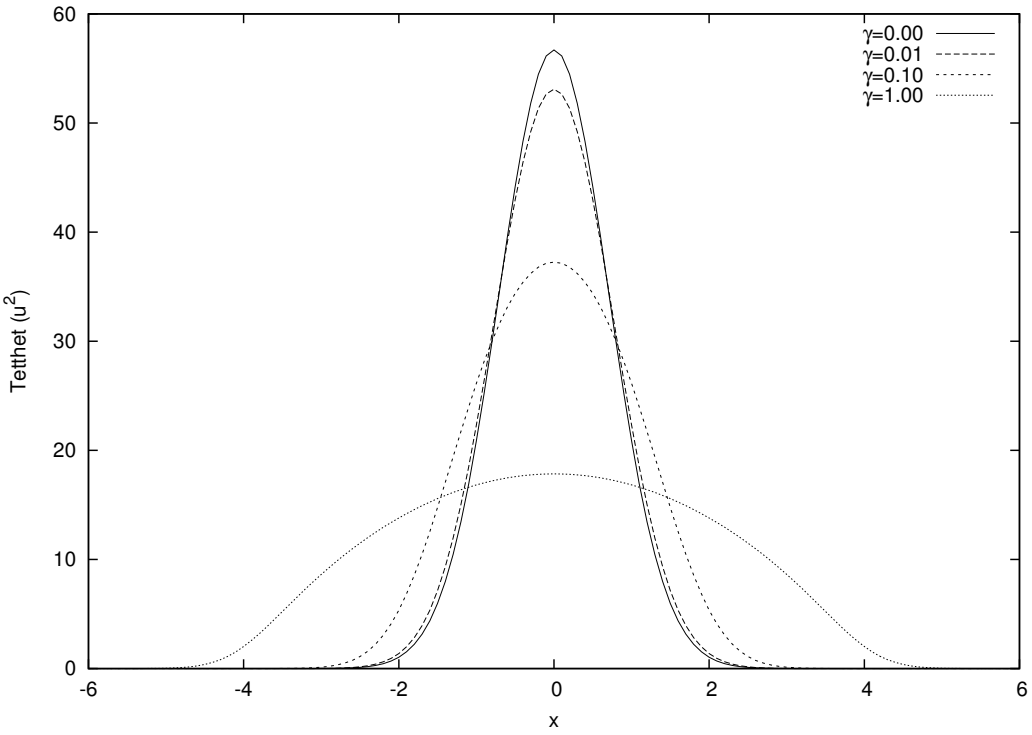
Siden det ikkelineære leddet er $\gamma|u|^2 = \gamma n|u|^2 / \int |u|^2 dx$, har økning av partikkeltallet akkurat samme effekt på egenverdien som økning av vekselvirkningen.



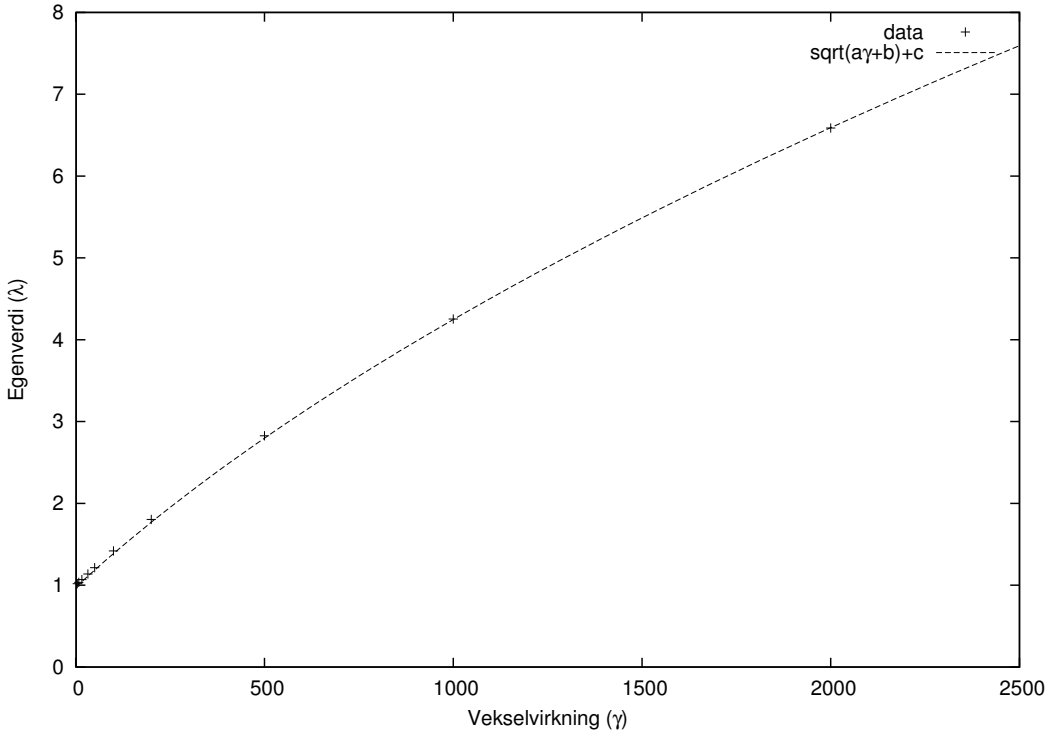
Figur 2: *Sammenligning av Newton-Raphson og den iterative metoden*



Figur 3: *Residual som funksjon av antall elementer*



Figur 4: *Tetthetsfordeling for 100 partikler ved forskjellige vekselvirkningsstyrker*



Figur 5: *Egenverdi (energi per partikkel) og en kurvetilpasning*

5 Implementasjon

5.1 Generelt om implementasjonen

Begge variantene arver FEM og NonLinSolver fra diffpack til å holde orden på FE-formulering av problemet og iterasjon over lineære løsninger. Men de tar kontroll over selve løsingen av det lineære systemet, for det gjør ikke diffpack for et egenverdiproblem.

BEC1d kutter vekk rader og kolonner som refererer til randelementer fra matrisene (og setter dermed randen til 0). Deretter sendes de til arpack som løser det lineære systemet i samarbeid med lineærløseren i diffpack.

Merk at denne fremgangsmåten avhenger av at randbetingelsen er 0. Det er greit i dette tilfellet, for tilstrekkelig langt fra sentrum bør tettheten være neglisjerbar.

NewtonBEC1d kutter matrisene på samme måte, og så konstrueres et nytt ligningssystem med en ekstra ukjent, som beskrevet i tidligere avsnitt. Dette løses av lineærløseren i diffpack. (Men kanskje ikke optimalt; en slags prekondisjonering el.l. bør sannsynligvis benyttes for å gjøre venstresiden gunstigere for løseren.)

Siden Newton-Raphson benyttes på matrisenivå er integrand-beregningen i FE-fasen identisk med den i BEC1d.

EigenSolver stammer fra Simen Kvaal, men med en del endringer. Den er et grensesnitt til arpack, som gjør det trivielt å kalle arpack hvis man bruker diffpack.

5.2 Svakheter

Den første prototypen (som ikke er inkludert her) forsøkte jeg å lage generell med hensyn til antall dimensjoner, kompleks eller reell løsning, og så videre. Men etterhvert ble det mindre tid. Så versjonen som er her bruker `NUMT` og `real` om hverandre, og virker bare for 1D reelle problemer med konstant gitteravstand.

Det er også manglende deling av kode mellom de to metodene, og spesielt Newtons metode lider av manglende optimalisering, ved unødig mye kopiering av matriser fordi det er tre forskjellige matrisestørrelser involvert (alle elementer, indre elementer, indre elementer pluss 1).

Men resultatene er gode.

5.3 Kode

5.3.1 main

main.cpp

```
#include <BEC1d.h>
#include <NewtonBEC1d.h>
//#include <Wave1.h>

int main(int argc, const char* argv[])
```

```
{
  initDiffpack(argc, argv);
  if (argc > 1 && argv[1][0] == 'n') {
    global_menu.init("Nonlinear Bose-Einstein Condensate in 1 dimension (Gross-Pitayes
    NewtonBEC1d simulator;
    global_menu.multipleLoop(simulator);
  } else {
    global_menu.init("Nonlinear Bose-Einstein Condensate in 1 dimension (Gross-Pitayes
    BEC1d simulator;
    global_menu.multipleLoop(simulator);
  }
  return 0;
}
```

5.3.2 BEC1d

BEC1d.h

```
#ifndef __BEC1d_h__
#define __BEC1d_h__

#include <FEM.h>                // includes FieldFE.h, GridFE.h
#include <Arrays_real.h>        // for MatDiag
#include <SaveSimRes.h>
#include <LinEqAdmFE.h>

#include <NonLinEqSolverUDC.h>
#include <NonLinEqSolver_prm.h>
#include <NonLinEqSolver.h>

#include "EigenSolver.h"

class BEC1d : public FEM, public NonLinEqSolverUDC
{
 protected:
  Handle(GridFE)     grid;    // finite element grid
  Handle(DegFreeFE)  dof;     // field <-> linear system mapping
  Handle(FieldFE)    u;       // primary unknown (eigenfunction)
  NUMT               lambda;  // primary unknown (eigenvalue)
  Vec(real)          linear_solution;
  Handle(SaveSimRes) database;

  real gamma;
  real num_particles;
  real initial_weight;
  real length; // length of grid

  MatBand(NUMT) K, M, A;
  EigenSolver eigensolver;

  virtual void integrands(ElmMatVec &elmat, const FiniteElement &fe);
  virtual real normOfResidual(Norm_type normt = L2);
```

```
35
    // extensions for nonlinear problems:
    Vec(real)                  nonlin_solution;
    Handle(NonLinEqSolver_prm) nlsolver_prm;
    Handle(NonLinEqSolver)     nlsolver;
40
    virtual void makeAndSolveLinearSystem();

   public:
    virtual void adm(MenuSystem& menu);
45  virtual void define(MenuSystem& menu, int level = MAIN);
    virtual void scan();

    virtual void solveProblem();
    virtual void resultReport();
50 };

  #endif
```

BEC1d.cpp

```
  #include "BEC1d.h"

  #include <readOrMakeGrid.h>
  #include <FiniteElement.h>
5 #include <ErrorNorms.h>

  void BEC1d::adm(MenuSystem &menu)
  {
    SimCase::attach(menu);
10  define(menu);
    menu.prompt();
    scan();
  }

15 void BEC1d::define(MenuSystem &menu, int level)
  {
    menu.addItem(level, "gridfile", "readOrMakeGrid syntax",
                 "P=PreproBox | d=1 [-10,10] | d=1 elm_tp=ElmB2n1D div=[50], grading=[1]

20  menu.addItem(level, "gamma",     "strength of interaction",     "0.0109");
    menu.addItem(level, "particles", "number of particles in trap", "1");

    menu.addItem(level, "tolerance", "EigenSolver tolerance",          "1e-5");
    menu.addItem(level, "per_iter",  "EigenSolver work per iteration", "15");
25  menu.addItem(level, "iterations","EigenSolver max iterations",     "1000");

    menu.addItem(level, "initial_w", "Use the analytical solution initially",
  "1.0");

    FEM::               defineStatic(menu, level+1);
30  NonLinEqSolver_prm::defineStatic(menu, level+1);
    SaveSimRes::        defineStatic(menu, level+1);
```

```
}

void BEC1d::scan()
{
  MenuSystem &menu = SimCase::getMenuSystem();
  String gridfile = menu.get("gridfile");
  grid.rebind(new GridFE());              // create empty grid object
  readOrMakeGrid(*grid, gridfile);        // fill grid

  Ptv(real) x0(1), x1(1);      // 1D specific
  grid->getMinMaxCoord(x0,x1);
  length = x1(1)-x0(1);

  database.rebind(new SaveSimRes());
  database->scan(menu, grid->getNoSpaceDim());
  FEM::scan(menu);

  gamma         = atof(menu.get("gamma").c_str());
  num_particles = atof(menu.get("particles").c_str());
  initial_weight= atof(menu.get("initial_w").c_str());

  real tol;
  tol = atof(menu.get("tolerance").c_str());
  eigensolver.setTolerance(tol);
  int num;
  num = atoi(menu.get("per_iter").c_str());
  eigensolver.setWorkPerIteration(num);
  num = atoi(menu.get("iterations").c_str());
  eigensolver.setMaxIterations(num);

  u.rebind(new FieldFE(*grid,"u"));             // allocate space for u
  dof.rebind(new DegFreeFE(*grid, 1));          // 1 unknown per node

  nlsolver_prm = NonLinEqSolver_prm::construct();
  nlsolver_prm->scan(menu);

  nlsolver.rebind(nlsolver_prm->create());
  nonlin_solution.redim(dof->getTotalNoDof());
  linear_solution.redim(dof->getTotalNoDof());
  nlsolver->attachUserCode(*this);
  nlsolver->attachNonLinSol(nonlin_solution);
  nlsolver->attachLinSol(linear_solution);

  A.redim(grid->getNoNodes(), grid->calcBandwidth(), false, false);
}

real analyticalSolution (const Ptv(NUMT)& x, real n)
{
  const real sqrt_pi = 1.77245385090552;
  return exp(-x.sqr()/2) * sqrt(n/sqrt_pi);
}

void BEC1d::solveProblem()
```

```
{
  //
  // Initial guess for u: use (weighted) analytical solution
  //

  const int no_nodes = grid->getNoNodes();
  Ptv(real) p(grid->getNoSpaceDim());

  for (int i = 1; i <= no_nodes; i++) {
    grid->getCoor(p, i);
    nonlin_solution(i) = initial_weight*analyticalSolution(p, num_particles);
  }

  //
  // Calculate RHS (the constant matrix M)
  //

  makeMassMatrix(*grid, A, false);
  eigensolver.enforceZeroEssBC(*grid, A, M);

  //
  // Call the nonlinear solver
  //

  bool converged = nlsolver->solve();

  // load nonlinear solution found by the solver into the u field:
  dof->vec2field(nonlin_solution, *u);

  if (!converged)
    errorFP("BEC1d::solveProblem",
            "The nonlinear solver \"%s\" did not fulfill the convergence\n"
            "criterion in %d iterations (final \"error\" estimate: eps=%g).",
            getEnumValue(nlsolver->getCurrentState().method).c_str(),
            nlsolver->getCurrentState().iteration_no,
            nlsolver->getCurrentState().eps);
}

void BEC1d::makeAndSolveLinearSystem ()
{
  dof->vec2field(nonlin_solution, *u); // copy most recent guess to u

  //
  // Re-calculate LHS (the current best guess at K(u))
  //

  makeSystem(*dof, A);
  eigensolver.enforceZeroEssBC(*grid, A, K);

  //
  // Attach matrices and call the eigensolver on
  //         K(u*) u = lambda M u
  //
```

```
    Vec(NUMT) eigenvalues;
140 Mat(NUMT) eigenvectors;

    eigensolver.attach(K, M, true);
    eigensolver.solve(1, eigenvalues, eigenvectors);

145 if (eigenvalues.size() == 0)
      errorFP("BEC1d::makeAndSolveLinearSystem",
              "Arpack didn't converge");

    eigensolver.setInitialVector(eigenvectors,1); // initial search for NEXT iteration
150
    //
    // Store the (linear solution) eigenvector & value
    //

155 eigensolver.extractEssBCEigenvector(*grid, eigenvectors, linear_solution, 1);
    lambda = eigenvalues(1);

    //
    // Re-normalise vector to be sqrt(density)
160 // NOTE assumes uniform 1D grid
    //

    int no_nodes = grid->getNoNodes();
    real sign = (linear_solution(no_nodes/2) < 0 ? -1.0 : 1.0);
165 real unscaled_num_particles = sqr(linear_solution.norm()) * length/no_nodes;

    for (int i=2; i<=no_nodes-1; i++)
      linear_solution(i) *= sign*sqrt(num_particles/unscaled_num_particles);
  }
170
  real BEC1d::normOfResidual(Norm_type normt)
  {
    int nno = grid->getNoNodes();
    Vec(real) residues(nno-2);
175 real dx2 = sqr(length/(nno-1));

    Ptv(real) p(1);

    for (int i = 2; i <= nno-1; i++) {
180   grid->getCoor(p,i);               // get the coord. p of node i

      real u_pt = u->valueNode(i);
      real ddu_pt = (u->valueNode(i-1) - 2*u_pt + u->valueNode(i+1)) / dx2;

185   residues(i-1) = norm(ddu_pt + (lambda - p.sqr() - gamma*sqr(u_pt)) * u_pt);
    }

    return residues.norm(normt);
  }
190
```

```
void BEC1d::integrands(ElmMatVec &elmat, const FiniteElement &fe)
{
  const int  nsd    = fe.getNoSpaceDim();    // no of space dim.
  const Ptv(real) x = fe.getGlobalEvalPt();  // x at present itg.pt.
  const NUMT u_pt   = u->valueFEM (fe);      // u at present itg.pt.
  const int nbf     = fe.getNoBasisFunc();   // no of nodes in this elm.
  const real detJxW = fe.detJxW();

  switch (nlsolver->getCurrentState().method) {

  case SUCCESSIVE_SUBST:
   for (int i = 1; i <= nbf; i++)
      for (int j = 1; j <= nbf; j++)  {

        real nabla2 = 0;
        for (int k = 1; k <= nsd; k++)
          nabla2 += fe.dN(i,k)*fe.dN(j,k);

        real pot_energy = fe.N(i) * fe.N(j) * (x.sqr() + gamma*norm(u_pt));

        elmat.A(i,j) += (nabla2 + pot_energy ) * detJxW;
      }

    break;

  default:
    errorFP("BEC1d::integrands",
            "Linear subsystem for the nonlinear method %s is not implemented",
            getEnumValue(nlsolver->getCurrentState().method).c_str());
  }
}

void BEC1d::resultReport()
{
  real L1_error, L2_error, Linf_error;
  ErrorNorms::Lnorm (analyticalSolution, // supplied function
                     *u,                 // numerical solution
                     num_particles,      // really meant for time
                     L1_error, L2_error, Linf_error, // error norms
                     NODAL_POINTS);      // use nodes only
  // since we will have exact solution at the nodes, the errors should be 0!
  s_o << "Difference from analytical solution with gamma=0:\n";
  s_o << oform("L1-error=%12.5e,  L2-error=%12.5e,  max-error=%12.5e\n\n",
              L1_error, L2_error, Linf_error);


  s_o << oform("lambda = %e  eps = %e\n", lambda,  nlsolver->getCurrentState().eps);
  database->curve(*u, NULL, NULL, aform("; lambda = %e", lambda).c_str());
}
```

5.3.3 NewtonBEC1d

```
#ifndef __NewtonBEC1d_h__
#define __NewtonBEC1d_h__

#include <FEM.h>                // includes FieldFE.h, GridFE.h
#include <Arrays_real.h>        // for MatDiag
#include <SaveSimRes.h>
#include <LinEqAdmFE.h>

#include <NonLinEqSolverUDC.h>
#include <NonLinEqSolver_prm.h>
#include <NonLinEqSolver.h>

class NewtonBEC1d : public FEM, public NonLinEqSolverUDC
{
protected:
  // general data:
  Handle(GridFE)     grid;    // finite element grid
  Handle(DegFreeFE)  dof;     // field <-> linear system mapping
  Handle(FieldFE)    u;       // primary unknown

  Vec(real)          linear_solution;
  Vec(real)          linear_with_lambda;
  real lambda;

  real num_particles;
  real gamma;
  real length;

  MatBand(NUMT) M, K;
  Mat(NUMT) A;
  Vec(NUMT) b;

  Handle(LinEqAdm) lineq;   // linear system & solution
  Handle(SaveSimRes) database;

  void extractSolution();
  void assembleLinearSystem();
  void setIC();
  virtual void integrands (ElmMatVec& elmat, const FiniteElement& fe);
  virtual real normOfResidual(Norm_type normt = L2);

  // extensions for nonlinear problems:
  Vec(real)                  nonlin_solution;
  Handle(NonLinEqSolver_prm) nlsolver_prm;
  Handle(NonLinEqSolver)     nlsolver;

  virtual void makeAndSolveLinearSystem ();

public:
  virtual void adm    (MenuSystem& menu);
  virtual void define (MenuSystem& menu, int level = MAIN);
```

```
    virtual void scan   ();

    virtual void solveProblem ();
    virtual void resultReport ();
};
#endif
```

NewtonBEC1d.cpp

```
#include <NewtonBEC1d.h>
#include "EigenSolver.h"

#include <ElmMatVec.h>
#include <FiniteElement.h>
#include <readOrMakeGrid.h>
#include <ErrorNorms.h>

void NewtonBEC1d::adm(MenuSystem& menu)
{
  SimCase::attach(menu);
  define(menu);
  menu.prompt();
  scan();
}

void NewtonBEC1d::define(MenuSystem& menu, int level)
{
  menu.addItem(level, "gridfile", "readOrMakeGrid syntax",
               "P=PreproBox | d=1 [-10,10] | d=1 elm_tp=ElmB2n1D "
               "div=[20], grading=[1]");
  menu.addItem(level, "particles", "number of particles in trap", "1");
  menu.addItem(level, "gamma", "strength of interaction", "0");

  // submenus:
  FEM::               defineStatic(menu, level+1);
  NonLinEqSolver_prm::defineStatic(menu, level+1);
  SaveSimRes::        defineStatic(menu, level+1);
  LinEqAdm::          defineStatic(menu, level+1);
}

void NewtonBEC1d::scan()
{
  MenuSystem& menu = SimCase::getMenuSystem();
  String gridfile = menu.get("gridfile");
  grid.rebind(new GridFE());              // create empty grid object
  readOrMakeGrid(*grid, gridfile);        // fill grid
  database.rebind(new SaveSimRes());
  database->scan(menu, grid->getNoSpaceDim());
  FEM::scan(menu);

  num_particles = atof(menu.get("particles").c_str());
  gamma = atof(menu.get("gamma").c_str());
```

```
  Ptv(real) x0(1), x1(1);      // 1D specific
  grid->getMinMaxCoord(x0,x1);
  length = x1(1)-x0(1);

  u.rebind(new FieldFE(*grid,"u"));             // allocate space for u
  dof.rebind(new DegFreeFE(*grid, 1));          // 1 unknown per node

  // create parameter objects this way:
  nlsolver_prm.rebind(NonLinEqSolver_prm::construct());
  nlsolver_prm->scan(menu);
  // create iteration method (subclass of NonLinEqSolver):
  nlsolver.rebind(nlsolver_prm->create());
  nonlin_solution.redim(dof->getTotalNoDof());
  linear_solution.redim(dof->getTotalNoDof());

  int interior_nodes = 0;
  for (int i=1; i<=grid->getNoNodes(); i++)
    if (!grid->boNode(i)) interior_nodes++;

  A.redim(interior_nodes+1, interior_nodes+1);
  A.fill(0);
  linear_with_lambda.redim(interior_nodes+1);
  b.redim(interior_nodes+1);

  lineq.rebind(new LinEqAdm(EXTERNAL_STORAGE));
  lineq->attach(A,linear_with_lambda,b);
  lineq->scan(menu);

  K.redim(grid->getNoNodes(), grid->calcBandwidth(), false, false);
  M.redim(grid->getNoNodes(), grid->calcBandwidth(), false, false);

  nlsolver->attachUserCode(*this);
  nlsolver->attachNonLinSol(nonlin_solution);
  nlsolver->attachLinSol(linear_solution);
  // all data structures are allocated
}

// implemented in BEC1d.cpp
real analyticalSolution(const Ptv(real)& x, real n);


void NewtonBEC1d::resultReport()
{
  real L1_error, L2_error, Linf_error;
  ErrorNorms::Lnorm(analyticalSolution, // supplied function
                    *u,                 // numerical solution
                    num_particles,      // really meant for time
                    L1_error, L2_error, Linf_error, // error norms
                    NODAL_POINTS);      // use nodes only
  // since we will have exact solution at the nodes, the errors should be 0!
  s_o << "Difference from analytical solution with gamma=0:\n";
  s_o << oform("L1-error=%12.5e,  L2-error=%12.5e,  max-error=%12.5e\n\n",
               L1_error, L2_error, Linf_error);
```

```
100   s_o << oform("lambda = %e  eps = %e\n", lambda,  nlsolver->getCurrentState().eps);
      database->curve(*u, NULL, NULL, aform("; lambda = %e", lambda).c_str());
    }

    void NewtonBEC1d::setIC()
105 {
      const int nno = grid->getNoNodes();
      Ptv(real) p(grid->getNoSpaceDim());
      for (int i = 1; i <= nno; i++) {
        grid->getCoor(p,i);               // get the coord. p of node i
110     u->valueNode(i) = analyticalSolution(p, 1); // assign nodal value
      }
    }

    void NewtonBEC1d::solveProblem()
115 {
      setIC();

      // initialize nonlin_solution with the analytical soln
      dof->field2vec(*u, nonlin_solution);
120   makeMassMatrix(*grid, M, false);

      // call nonlinear solver:
      bool converged = nlsolver->solve();

125   // load nonlinear solution found by the solver into the u field:
      dof->vec2field(nonlin_solution, *u);

      if (!converged)
        errorFP("NewtonBEC1d::solveAtThisTimeStep",
130             "The nonlinear solver \"%s\" did not fulfill the convergence\n"
                "criterion in %d iterations (final \"error\" estimate: eps=%g).",
                getEnumValue(nlsolver->getCurrentState().method).c_str(),
                nlsolver->getCurrentState().iteration_no,
                nlsolver->getCurrentState().eps);
135 }

    void NewtonBEC1d::assembleLinearSystem()
    {
      MatBand(NUMT) K_reduced, M_reduced;
140   EigenSolver::enforceZeroEssBC(*grid, K, K_reduced);
      EigenSolver::enforceZeroEssBC(*grid, M, M_reduced);

      int n = K_reduced.getNoRows(); // number of interior nodes, without lambda

145   Vec(NUMT) u_reduced(n);

      for (int i=1, j=1; i<=grid->getNoNodes(); i++)
        if (!grid->boNode(i))
          u_reduced(j++) = nonlin_solution(i);
150
```

```
    Vec(NUMT) Ku(n); K_reduced.prod(u_reduced, Ku);
    Vec(NUMT) Mu(n); M_reduced.prod(u_reduced, Mu);

    //     [  A - lambda B    -B x  ]
155 // A = |                         |
    //     |     T                   |
    //     [  2 x              0     ]

    for (int i=1; i<=n; i++) {
160   for (int j=1; j<=n; j++) {
        A(i,j) = 0.0;
        if (K_reduced.insideBand(i,j))
          A(i,j) += K_reduced.elm(i,j);
        if (M_reduced.insideBand(i,j))
165       A(i,j) -= lambda*M_reduced.elm(i,j);
      }
      A(i,n+1) = -Mu(i);
      A(n+1,i) = 2*u_reduced(i);
    }
170 A(n+1,n+1) = 0;

    //       [  (A - lambda B) x  ]
    // b = - |                      |
    //       |       T              |
175 //       [      x  x - c       ]

    for (int i=1; i<=n; i++)
      b(i) = lambda*Mu(i) - Ku(i);
    b(n+1) = num_particles*grid->getNoNodes()/length - sqr(u_reduced.norm());
180 }

  void NewtonBEC1d::extractSolution()
  {
    int j=1;
185 for (int i=1; i<=grid->getNoNodes(); i++)
      if (grid->boNode(i))
        linear_solution(i) = 0;
      else {
        linear_solution(i) = linear_with_lambda(j);
190     j++;
      }

    lambda += linear_with_lambda(j);
  }
195
  void NewtonBEC1d::makeAndSolveLinearSystem()
  {
    dof->vec2field(nonlin_solution, *u); // copy most recent guess to u

200 makeSystem(*dof, K);
    assembleLinearSystem();

    lineq->solve();
```

```
  extractSolution();
}

real NewtonBEC1d::normOfResidual(Norm_type normt)
{
  int nno = grid->getNoNodes();
  Vec(real) residues(nno-2);
  real dx2 = sqr(length/(nno-1));

  Ptv(real) p(1);

  for (int i = 2; i <= nno-1; i++) {
    grid->getCoor(p,i);               // get the coord. p of node i

    real u_pt = u->valueNode(i);
    real ddu_pt = (u->valueNode(i-1) - 2*u_pt + u->valueNode(i+1)) / dx2;

    residues(i-1) = norm(ddu_pt + (lambda - p.sqr() - gamma*sqr(u_pt)) * u_pt);
  }

  return residues.norm(normt);
}

void NewtonBEC1d::integrands(ElmMatVec& elmat,const FiniteElement& fe)
{
  const int  nsd    = fe.getNoSpaceDim();    // no of space dim.
  const Ptv(real) x = fe.getGlobalEvalPt();  // x at present itg.pt.
  const NUMT u_pt   = u->valueFEM(fe);      // u at present itg.pt.
  const int nbf     = fe.getNoBasisFunc();   // no of nodes in this elm.
  const real detJxW = fe.detJxW();

  switch (nlsolver->getCurrentState().method) {

  case NEWTON_RAPHSON:
    for (int i = 1; i <= nbf; i++)
      for (int j = 1; j <= nbf; j++)  {

        real nabla2 = 0;
        for (int k = 1; k <= nsd; k++)
          nabla2 += fe.dN(i,k)*fe.dN(j,k);

        real pot_energy = fe.N(i) * fe.N(j) * (x.sqr() + gamma*norm(u_pt));

        elmat.A(i,j) += (nabla2 + pot_energy ) * detJxW;
      }

    break;

  default:
    errorFP("BEC1d::integrands",
            "Linear subsystem for the nonlinear method %s is not implemented",
            getEnumValue(nlsolver->getCurrentState().method).c_str());
```

```
  }
}
```

5.3.4 EigenSolver

EigenSolver.h

```
#ifndef __EigenSolver_h__
#define __EigenSolver_h__

#include <FEM.h>
#include <LinEqAdm.h>

enum Mode { REGULAR_MODE, SHIFT_INVERT_MODE, CAYLEY_MODE, BUCKLING_MODE, COMPLEX_SHIFT

enum Spectrum { SMALLEST_MAGNITUDE, SMALLEST_ALGEBRAIC, SMALLEST_REAL, SMALLEST_IMAG,
                LARGEST_MAGNITUDE,  LARGEST_ALGEBRAIC,  LARGEST_REAL,  LARGEST_IMAG,
                BOTH_ENDS };

class EigenSolver
{
 public:
  EigenSolver();
  ~EigenSolver();

  void attach(Matrix(NUMT) &A,                   bool assumeSymmetric = false);
  void attach(Matrix(NUMT) &A, Matrix(NUMT) &B, bool assumeSymmetric = false);

  static void enforceZeroEssBC(const GridFE &grid, const MatBand(NUMT) &A, MatBand(NUM
  static void extractEssBCEigenvector(const GridFE &grid, const Mat(NUMT) &A, Vec(NUMT

  void setShift(NUMT sigma, Mode = SHIFT_INVERT_MODE);
  void setComplexShift(real sigma_re, real sigma_im); // ONLY for real, non-symmetric,
  void setSpectrum(Spectrum part);

  void setInitialVector(Vec(NUMT) &vec);
  void setInitialVector(Mat(NUMT) &mat, int n);

  void setTolerance(real tol);
  void setMaxIterations(int max_it);
  void setWorkPerIteration(int n_cv);

  void solve(int nev, Vec(NUMT) &values, Mat(NUMT) &vectors);

 private:
  void setDefaults();

  void multAx(NUMT *, NUMT *);
  void multBx(NUMT *, NUMT *);
  void multInvCDx(NUMT *, NUMT *);
  void multInvCx(NUMT *, NUMT *);

```

```
    NUMT sigma;
    real sigma_im;
    Mode mode;
    Spectrum spectrum;

    int n_per_iter;
    real tolerance;
    int max_iter;
    NUMT *initial_vector;

    bool is_real_symmetric;
    bool is_generalized;
    bool is_complex;

    int size;

    Handle(Matrix(NUMT)) A, B;     // problem defining matrices
    Handle(Vec(NUMT)) V, W;        // aux vectors for matrix-vector operations

    // linear equations objects.
    Handle(Matrix(NUMT)) C, D;
    Handle(LinEqSolver) linear_solver;
    Handle(LinEqSolver_prm) linear_solver_prm;
    Handle(LinEqAdm) lineq;
    bool C_has_changed;  // indicates whether we do the first linear solve or not

    Handle(MenuSystem) menu;
};

#endif
```

EigenSolver.cpp

```
#include "EigenSolver.h"

#include <Arrays_real.h>

#if NUMT_IS_COMPLEX == 1
#include <argcomp.h>
#define arNUMT arcomplex<real>
#define is_complex true
#else
#include <argsym.h>
#define arNUMT real
#define is_complex false
#endif

void EigenSolver::setDefaults()
{
  this->mode = REGULAR_MODE;
  this->sigma = 0;
  this->sigma_im = 0;
  this->spectrum = SMALLEST_MAGNITUDE;
```

```
    this->n_per_iter = 0;       // -> (size+n_ev)/2
    this->tolerance = 0.0;      // -> machine precision
    this->max_iter = 0;         // -> 100
}

EigenSolver::EigenSolver()
{
  linear_solver_prm.rebind( LinEqSolver_prm::construct() );
  //if (menu_handle != NULL)
  //  linear_solver_prm->scan(*menu_handle);
  linear_solver.rebind( linear_solver_prm->create() );

  lineq.rebind( new LinEqAdm(EXTERNAL_STORAGE) );
  lineq->attach( linear_solver() );

  setDefaults();

  V.rebind( new Vec(NUMT)(0) );
  W.rebind( new Vec(NUMT)(0) );
  initial_vector = NULL;
}


void EigenSolver::attach(Matrix(NUMT) &A, bool assume_symmetric)
{
  this->is_real_symmetric = (!is_complex && (assume_symmetric || A.symmetricStorage())
  this->is_generalized = false;
  this->A.rebind(A);

  if (size != A.getNoColumns() && initial_vector) {
    delete[] initial_vector;
    initial_vector = NULL;
  }

  this->size = A.getNoColumns();
  this->C_has_changed = true;
}


void EigenSolver::attach(Matrix(NUMT) &A, Matrix(NUMT) &B, bool assume_symmetric)
{
  if (A.getNoColumns() != B.getNoColumns())
    errorFP("EigenSolver::attach",
            "Matrices A and B are of different size");

  this->attach(A, assume_symmetric);
  this->is_real_symmetric = (is_real_symmetric && (assume_symmetric || B.symmetricStor
  this->is_generalized = true;
  this->B.rebind(B);
}
```

```
void EigenSolver::setShift(NUMT sigma, Mode mode)
{
  static char *where = "EigenSolver::setShift";

  switch(mode) {
  case REGULAR_MODE:
    if (sigma != 0.0)
      warningFP(where, "Regular mode with sigma!=0; ignoring sigma");
    break;
  case CAYLEY_MODE: case BUCKLING_MODE:
    if (!is_real_symmetric || !is_generalized || sigma == 0.0)
      errorFP(where, "Requested mode is only for real symmetric generalized problems w
    break;
  case COMPLEX_SHIFT_MODE:
    errorFP(where, "Use the setShift(real sigma_re, real sigma_im) method to set compl
  }

  this->sigma = sigma;
  this->mode  = mode;
}


void EigenSolver::setComplexShift(real sigma_re, real sigma_im)
{
  if (is_complex || is_real_symmetric || !is_generalized || (sigma_re == 0.0 && sigma_
    errorFP("EigenSolver::setComplexShift",
            "Only for real, non-symmetric, generalized problems with |sigma|>0");

  this->sigma = sigma_re;
  this->sigma_im = sigma_im;
  this->mode = COMPLEX_SHIFT_MODE;
}


void EigenSolver::setSpectrum(Spectrum part)
{
  static char *where = "EigenSolver::setSpectrum";

  switch (part) {
  case SMALLEST_REAL: case LARGEST_REAL: case SMALLEST_IMAG: case LARGEST_IMAG:
    if (is_real_symmetric)
      errorFP(where, "Requested spectrum is only for non-symmetric or complex matrices
    break;
  case LARGEST_ALGEBRAIC: case SMALLEST_ALGEBRAIC: case BOTH_ENDS:
    if (!is_real_symmetric)
      errorFP(where, "Requested spectrum is only for real symmetric matrices");
  }

  this->spectrum = part;
}


void EigenSolver::solve(int n_ev, Vec(NUMT) &out_values, Mat(NUMT) &out_vectors)
```

```
  {
    // missing: non-symmetric real, complex shift-invert,
    static char *spectrum_array[] = { "SM", "SA", "SR", "SI", "LM", "LA", "LR", "LI", "B
130 char *spectrum_str = spectrum_array[spectrum];

    arNUMT *resid = (arNUMT *)initial_vector;

    if (n_ev > size-1) n_ev = size-1;
135 if (n_ev < 1)      n_ev = 1;

    n_per_iter = (size+n_ev)/2;

    typedef void (EigenSolver::*multfunc)(arNUMT*, arNUMT *);
140
    ARrcStdEig<real,arNUMT> *solver = NULL;

  #if NUMT_IS_COMPLEX == 1
    {
145   if (!is_generalized && mode == REGULAR_MODE)
        solver = new ARCompStdEig<real,EigenSolver>(size, n_ev,
                                                    this, (multfunc)&EigenSolver::multAx,
                                                    spectrum_str, n_per_iter,
                                                    tolerance, max_iter, resid);
150   if (is_generalized && mode == REGULAR_MODE) {
        D.rebind(*A);
        B->makeItSimilar(C);
        *C=*B;
        solver = new ARCompGenEig<real,EigenSolver,EigenSolver>(size, n_ev,
155                                                             this, (multfunc)&EigenSolver
                                                                this, (multfunc)&EigenSolver
                                                                spectrum_str, n_per_iter,
                                                                tolerance, max_iter, resid);
      }
160 }
  #else // ie. NUMT is real
    {
      if (is_real_symmetric) {
        if (!is_generalized) {
165       if (mode == REGULAR_MODE)
            solver = new ARSymStdEig<real,EigenSolver>(size, n_ev,
                                                       this, (multfunc)&EigenSolver::multA
                                                       spectrum_str, n_per_iter,
                                                       tolerance, max_iter, resid);
170       if (mode == SHIFT_INVERT_MODE) {
            A->makeItSimilar(C);
            *C=*A;
            for (int i=1; i<=size; i++)
              C->elm(i,i) -= sigma; // C  <-  A - sigma * I
175         solver = new ARSymStdEig<real,EigenSolver>(size, n_ev,
                                                       this, (multfunc)&EigenSolver::multI
                                                       sigma, spectrum_str, n_per_iter,
                                                       tolerance, max_iter, resid);
          }
```

```
180       }
          else { //generalized
            if (mode == REGULAR_MODE) {
              D.rebind(*A);
              B->makeItSimilar(C);
185           *C = *B;
              solver = new ARSymGenEig<real,EigenSolver,EigenSolver>(size, n_ev,
                                                                     this, (multfunc)&EigenSol
                                                                     this, (multfunc)&EigenSol
                                                                     spectrum_str, n_per_iter,
190                                                                  tolerance, max_iter, resi
            }
            if (mode == SHIFT_INVERT_MODE || mode == BUCKLING_MODE || mode == CAYLEY_MODE)
              B->makeItSimilar(C);
              add(*C,*A,-sigma,*B); // C  <-  A - sigma * B
195         }
            if (mode == SHIFT_INVERT_MODE)
              solver = new ARSymGenEig<real,EigenSolver,EigenSolver>('S', size, n_ev,
                                                                     this, (multfunc)&EigenSol
                                                                     this, (multfunc)&EigenSol
200                                                                  sigma, spectrum_str, n_pe
                                                                     tolerance, max_iter, resi

            if (mode == BUCKLING_MODE)
              solver = new ARSymGenEig<real,EigenSolver,EigenSolver>('B', size, n_ev,
205                                                                  this, (multfunc)&EigenSol
                                                                     this, (multfunc)&EigenSol
                                                                     sigma, spectrum_str, n_pe
                                                                     tolerance, max_iter, resi

210         if (mode == CAYLEY_MODE)
              solver = new ARSymGenEig<real,EigenSolver,EigenSolver>(size, n_ev,
                                                                     this, (multfunc)&EigenSol
                                                                     this, (multfunc)&EigenSol
                                                                     this, (multfunc)&EigenSol
215                                                                  sigma, spectrum_str, n_pe
                                                                     tolerance, max_iter, resi
          }
        }
      }
220 #endif

    if (!solver)
      errorFP("EigenSolver::solve()",
              "Couldn't create solver (probably because it isn't implemented)");
225
    solver->FindEigenvectors();

    void *eval_ptr = (void *)solver->RawEigenvalues();
    void *evec_ptr = (void *)solver->RawEigenvectors();
230 int n_converged = solver->ConvergedEigenvalues();

    out_values.redim(n_converged);
```

```
    out_vectors.redim(n_converged, size);

    for (int i=0; i<n_converged; i++)
      out_values(i+1) = ((NUMT *)eval_ptr)[i];

    for (int i=0; i<n_converged; i++)
      for (int j=0; j<size; j++)
        out_vectors(i+1, j+1) = ((NUMT *)evec_ptr)[size*i + j];

    delete solver;
  }


  // matrix-vector multiplication: w  <-  A * v
  void EigenSolver::multAx(NUMT *v, NUMT *w)
  {
    V->redim(v, size);
    W->redim(w, size);

    A->prod(*V, *W);  // w  <-  A * v
  }

  // matrix-vector multiplication: w  <-  B * v
  void EigenSolver::multBx(NUMT *v, NUMT *w)
  {
    V->redim(v, size);
    W->redim(w, size);

    B->prod(*V, *W);  // w  <-  B * v
  }


  // matrix-vector multipliction: w  <-  inv(C) * D * v
  void EigenSolver::multInvCDx(NUMT *v, NUMT *w)
  {
    V->redim(v, size);
    W->redim(w, size);

    D->prod(*V, *W);             // w  <-  D * v

    *V = *W;                     // v  <-  w

    lineq->attach(*C, *W, *V);   // w  <-  inv(C) * v
    lineq->solve(C_has_changed);
    C_has_changed = false;
  }


  // matrix-vector multipliction: w  <-  inv(C) * v
  void EigenSolver::multInvCx(NUMT *v, NUMT *w)
  {
    V->redim(v, size);
    W->redim(w, size);
```

```
    lineq->attach(*C, *W, *V);  // w  <-  inv(C) * v
    lineq->solve(C_has_changed);
    C_has_changed = false;
}


  void EigenSolver::setTolerance(real tol)           { this->tolerance = tol; }
  void EigenSolver::setMaxIterations(int max_it)     { this->max_iter = max_it; }
void EigenSolver::setWorkPerIteration(int n_cv)    { this->n_per_iter = n_cv; }

  void EigenSolver::setInitialVector(Vec(NUMT) &vec)
  {
    if (vec.size() != this->size)
      errorFP("EigenSolver::setInitialVector()",
              "initial vector of wrong size");

    if (!initial_vector)
      initial_vector = new NUMT[size];

    for (int i=0; i<size; i++)
      initial_vector[i] = vec(i+1);
  }

void EigenSolver::setInitialVector(Mat(NUMT) &mat, int n)
  {
    if (mat.getNoColumns() != this->size)
      errorFP("EigenSolver::setInitialVector()",
              "Initial vector of wrong size");

    if (!initial_vector)
      initial_vector = new NUMT[size];

    for (int i=0; i<size; i++)
      initial_vector[i] = mat(n,i+1);
  }

  void EigenSolver::enforceZeroEssBC(const GridFE &grid,
                                     const MatBand(NUMT) &_A, MatBand(NUMT) &B)
{
    MatBand(NUMT) &A = (MatBand(NUMT)&)_A;  // discard const
    const int size = A.getNoRows();
    const int bw = A.getBandwidth();

    int nodesToCut = 0;
    for (int i=1; i<=size; i++)
      if (grid.boNode(i)) nodesToCut++;

    B.redim(size-nodesToCut, bw, A.symmetricStorage(), A.pivotingAllowed());
    B.fill(0.0);

    for (int A_r=1, B_r=1;  A_r <= size;  A_r++)
      if (!grid.boNode(A_r)) {
```

```
        // go right
        for (int A_e=A_r, B_e=B_r;  (A_e < A_r+bw) && (A_e <= size);  A_e++)
          if (!grid.boNode(A_e)) {
            B.elm(B_r,B_e) = A.elm(A_r,A_e);
            B_e++;
          }
        // go left
        for (int A_e=A_r-1, B_e=B_r-1;  (A_e > A_r-bw) && (A_e >= 1); A_e--)
          if (!grid.boNode(A_e)) {
            B.elm(B_r,B_e) = A.elm(A_r,A_e);
            B_e--;
          }
        B_r++;
      }
  }

void EigenSolver::extractEssBCEigenvector(const GridFE &grid,
                                          const Mat(NUMT) &_A, Vec(NUMT) &b, int n)
  {
    Mat(NUMT) &A = (Mat(NUMT) &)_A;   // discard const
    int no_nodes = grid.getNoNodes();

    b.redim(no_nodes); b.fill(0);

    for (int i=1, j=1; i<=no_nodes; i++)
      if (!grid.boNode(i)) {
        b(i) = A(n,j);
        j++;
      }
  }

EigenSolver::~EigenSolver() {
    if (initial_vector)
      delete[] initial_vector;
  }
```