Conjugate gradient methods and other optimization methods

Morten Hjorth-Jensen, National Superconducting Cyclotron Laboratory and Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, USA & Department of Physics, University of Oslo, Oslo, Norway

Spring 2015

Conjugate gradient (CG) method

Our aim with this part of the project is to be able to

- find an optimal value for the variational parameters using only some few Monte Carlo cycles
- use these optimal values for the variational parameters to perform a large-scale Monte Carlo calculation

To achieve this will look at methods like *Steepest descent* and the *conjugate gradient method*. Both these methods allow us to find the minima of a multivariable function like our energy (function of several variational parameters).

The success of the CG method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{A}\hat{x} = \hat{b}.$$

In the iterative process we end up with a problem like

$$\hat{r} = \hat{b} - \hat{A}\hat{x},$$

where $\hat{r}$ is the so-called residual or error in the iterative process.

Conjugate gradient method

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{b},$$

with the constraint that the matrix $\hat{A}$ is positive definite and symmetric. If we search for a minimum of the quantum mechanical variance, then the matrix $\hat{A}$, which is called the Hessian, is given by the second-derivative of the variance. This quantity is always positive definite.

Simple example and demonstration

Let us illustrate what is needed in our calculations using a simple example, the harmonic oscillator in one dimension. For the harmonic oscillator in one-dimension we have a trial wave function and probability

$$\psi_T(x) = e^{-\alpha^2 x^2} \qquad P_T(x)dx = \frac{e^{-2\alpha^2 x^2}dx}{\int dx e^{-2\alpha^2 x^2}}$$

with α being the variational parameter. We obtain then the following local energy

$$E_L[\alpha] = \alpha^2 + x^2\left(\frac{1}{2} - 2\alpha^2\right),$$

which results in the expectation value for the local energy

$$\langle E_L[\alpha]\rangle = \frac{1}{2}\alpha^2 + \frac{1}{8\alpha^2}$$

Repeat these calculations and convince yourself that they are correct.

Simple example and demonstration

The derivative of the energy with respect to α gives

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = \alpha - \frac{1}{4\alpha^3}$$

and a second derivative which is always positive (meaning that we find a minimum)

$$\frac{d^2\langle E_L[\alpha]\rangle}{d\alpha^2} = 1 + \frac{3}{4\alpha^4}$$

The condition

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 0,$$

gives the optimal $\alpha = 1/\sqrt{2}$, as expected. Show this.

Variance in the simple model

We can also minimize the variance. In our simple model the variance is

$$\sigma^2[\alpha] = \frac{1}{2}\alpha^4 - \frac{1}{4} + \frac{1}{32\alpha^4},$$

with first derivative

$$\frac{d\sigma^2[\alpha]}{d\alpha} = 2\alpha^3 - \frac{1}{8\alpha^5}$$

and a second derivative which is always positive

$$\frac{d^2\sigma^2[\alpha]}{d\alpha^2} = 6\alpha^2 + \frac{5}{8\alpha^6}$$

Computing the derivatives

In general we end up computing the expectation value of the energy in terms of some parameters $\alpha_0, \alpha_1, \ldots, \alpha_n$ and we search for a minimum in this multi-variable parameter space. This leads to an energy minimization problem *where we need the derivative of the energy as a function of the variational parameters.*

In the above example this was easy and we were able to find the expression for the derivative by simple derivations. However, in our actual calculations the energy is represented by a multi-dimensional integral with several variational parameters. How can we can then obtain the derivatives of the energy with respect to the variational parameters without having to resort to expensive numerical derivations?

Expressions for finding the derivatives of the local energy

To find the derivatives of the local energy expectation value as function of the variational parameters, we can use the chain rule and the hermiticity of the Hamiltonian.

Let us define

$$\bar{E}_\alpha = \frac{d\langle E_L[\alpha]\rangle}{d\alpha}.$$

as the derivative of the energy with respect to the variational parameter α (we limit ourselves to one parameter only). In the above example this was easy and we obtain a simple expression for the derivative. We define also the derivative of the trial function (skipping the subindex T) as

$$\bar{\psi}_\alpha = \frac{d\psi[\alpha]\rangle}{d\alpha}.$$

Derivatives of the local energy

The elements of the gradient of the local energy are then (using the chain rule and the hermiticity of the Hamiltonian)

$$\bar{E}_\alpha = 2\left(\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}E_L[\alpha]\rangle - \langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle\right).$$

From a computational point of view it means that you need to compute the expectation values of

$$\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}E_L[\alpha]\rangle,$$

and

$$\langle\frac{\bar{\psi}_\alpha}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle$$

We leave it as an exercise to find the corresponding expression for the variance.

Conjugate gradient method, Newton's method

In Newton's method we set $\nabla f = 0$ and we can thus compute the next iteration point (here the exact result)

$$\hat{x} - \hat{x}_i = \hat{A}^{-1}\nabla f(\hat{x}_i).$$

Subtracting this equation from that of $\hat{x}_{i+1}$ we have

$$\hat{x}_{i+1} - \hat{x}_i = \hat{A}^{-1}(\nabla f(\hat{x}_{i+1}) - \nabla f(\hat{x}_i)).$$

Concerning the optimization process in connection with the last project, you can easily use Newton's method as well instead of the con

Simple example and demonstration

In our case f can be either the energy or the variance. If we choose the energy then we have

$$\hat{\alpha}_{i+1} - \hat{\alpha}_i = \hat{A}^{-1}(\nabla E(\hat{\alpha}_{i+1}) - \nabla E(\hat{\alpha}_i)).$$

In the simple model gradient and the Hessian $\hat{A}$ are

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = \alpha - \frac{1}{4\alpha^3}$$

and a second derivative which is always positive (meaning that we find a minimum)

$$\hat{A} = \frac{d^2\langle E_L[\alpha]\rangle}{d\alpha^2} = 1 + \frac{3}{4\alpha^4}$$

Simple example and demonstration

We get then

$$\alpha_{i+1} = \frac{4}{3}\alpha_i - \frac{\alpha_i^4}{3\alpha_{i+1}^3},$$

which can be rewritten as

$$\alpha_{i+1}^4 - \frac{4}{3}\alpha_i\alpha_{i+1}^4 + \frac{1}{3}\alpha_i^4.$$

Using the conjugate gradient method

- Start your program with calling a function which implements for example the CGM method.
- This function needs the function for the expectation value of the local energy and the derivative of the local energy.
- Your function **func** is now the Metropolis part with a call to the local energy function. For every call to the function **func** many practitionser use 1000-10000 Monte Carlo cycles for the trial wave function.
- This gives an expectation value for the energy which is returned by the function **func**.
- When one calls the local energy one also computes the first derivative of the expectaction value of the local energy

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 2\langle \frac{\bar{\psi_{T\alpha}}}{\psi_T[\alpha]}\left(E_L[\alpha] - \langle E_L[\alpha]\rangle\right)\rangle.$$

Using the conjugate gradient method

The expectation value for the local energy of the Helium atom with a simple Slater determinant is given by

$$\langle E_L\rangle = \alpha^2 - 2\alpha\left(Z - \frac{5}{16}\right)$$

When implementing the conjugate gradient method, uyou should test your numerical derivative with the derivative of the last expression, that is

$$\frac{d\langle E_L[\alpha]\rangle}{d\alpha} = 2\alpha - 2\left(Z - \frac{5}{16}\right).$$

Conjugate gradient method

In the CG method we define so-called conjugate directions and two vectors $\hat{s}$ and $\hat{t}$ are said to be conjugate if

$$\hat{s}^T\hat{A}\hat{t} = 0.$$

The philosophy of the CG method is to perform searches in various conjugate directions of our vectors $\hat{x}_i$ obeying the above criterion, namely

$$\hat{x}_i^T\hat{A}\hat{x}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\hat{s}$ is conjugate to $\hat{t}$, then $\hat{t}$ is conjugate to $\hat{s}$.

Conjugate gradient method

An example is given by the eigenvectors of the matrix

$$\hat{v}_i^T \hat{A} \hat{v}_j = \lambda \hat{v}_i^T \hat{v}_j,$$

which is zero unless $i = j$.

Conjugate gradient method

Assume now that we have a symmetric positive-definite matrix $\hat{A}$ of size $n \times n$. At each iteration $i + 1$ we obtain the conjugate direction of a vector

$$\hat{x}_{i+1} = \hat{x}_i + \alpha_i \hat{p}_i.$$

We assume that $\hat{p}_i$ is a sequence of n mutually conjugate directions. Then the $\hat{p}_i$ form a basis of R^n and we can expand the solution $\hat{A}\hat{x} = \hat{b}$ in this basis, namely

$$\hat{x} = \sum_{i=1}^{n} \alpha_i \hat{p}_i.$$

Conjugate gradient method

The coefficients are given by

$$\mathbf{Ax} = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{p}_i = \mathbf{b}.$$

Multiplying with $\hat{p}_k^T$ from the left gives

$$\hat{p}_k^T \hat{A} \hat{x} = \sum_{i=1}^{n} \alpha_i \hat{p}_k^T \hat{A} \hat{p}_i = \hat{p}_k^T \hat{b},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\hat{p}_k^T \hat{b}}{\hat{p}_k^T \hat{A} \hat{p}_k}$$

Conjugate gradient method and iterations

If we choose the conjugate vectors $\hat{p}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\hat{x}$. So, we want to regard the conjugate gradient method as an iterative method. This also allows us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for $\hat{x}$ as $\hat{x}_0$. We can assume without loss of generality that

$$\hat{x}_0 = 0,$$

or consider the system

$$\hat{A}\hat{z} = \hat{b} - \hat{A}\hat{x}_0,$$

instead.

Conjugate gradient method

Important, one can show that the solution $\hat{x}$ is also the unique minimizer of the quadratic form

$$f(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{x}, \quad \hat{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{p}_1$ to be the gradient of f at $\hat{x} = \hat{x}_0$, which equals

$$\hat{A}\hat{x}_0 - \hat{b},$$

and $\hat{x}_0 = 0$ it is equal $-\hat{b}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Conjugate gradient method

Let $\hat{r}_k$ be the residual at the k-th step:

$$\hat{r}_k = \hat{b} - \hat{A}\hat{x}_k.$$

Note that $\hat{r}_k$ is the negative gradient of f at $\hat{x} = \hat{x}_k$, so the gradient descent method would be to move in the direction $\hat{r}_k$. Here, we insist that the directions $\hat{p}_k$ are conjugate to each other, so we take the direction closest to the gradient $\hat{r}_k$ under the conjugacy constraint. This gives the following expression

$$\hat{p}_{k+1} = \hat{r}_k - \frac{\hat{p}_k^T\hat{A}\hat{r}_k}{\hat{p}_k^T\hat{A}\hat{p}_k}\hat{p}_k.$$

Conjugate gradient method

We can also compute the residual iteratively as

$$\hat{r}_{k+1} = \hat{b} - \hat{A}\hat{x}_{k+1},$$

which equals

$$\hat{b} - \hat{A}(\hat{x}_k + \alpha_k\hat{p}_k),$$

or

$$(\hat{b} - \hat{A}\hat{x}_k) - \alpha_k\hat{A}\hat{p}_k,$$

which gives

$$\hat{r}_{k+1} = \hat{r}_k - \hat{A}\hat{p}_k,$$

Conjugate gradient method, our case

If we consider finding the minimum of a function f using Newton's method, that is search for a zero of the gradient of a function. Near a point x_i we have to second order

$$f(\hat{x}) = f(\hat{x}_i) + (\hat{x} - \hat{x}_i)\nabla f(\hat{x}_i)\frac{1}{2}(\hat{x} - \hat{x}_i)\hat{A}(\hat{x} - \hat{x}_i)$$

giving

$$\nabla f(\hat{x}) = \nabla f(\hat{x}_i) + \hat{A}(\hat{x} - \hat{x}_i).$$

In Newton's method we set $\nabla f = 0$ and we can thus compute the next iteration point (here the exact result)

$$\hat{x} - \hat{x}_i = \hat{A}^{-1}\nabla f(\hat{x}_i).$$

Subtracting this equation from that of $\hat{x}_{i+1}$ we have

$$\hat{x}_{i+1} - \hat{x}_i = \hat{A}^{-1}(\nabla f(\hat{x}_{i+1}) - \nabla f(\hat{x}_i)).$$

Simple codes for steepest descent and conjugate gradient

```
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "vectormatrixclass.h"
using namespace  std;
//   Main function begins here
int main(int  argc, char * argv[]){
  int dim = 2;
  Vector x(dim),xsd(dim), b(dim),x0(dim);
  Matrix A(dim,dim);

  // Set our initial guess
  x0(0) = x0(1) = 0;
  // Set the matrix
  A(0,0) =  3;    A(1,0) =  2;   A(0,1) =  2;   A(1,1) =  6;
  b(0) = 2; b(1) = -8;
```

Simple codes for steepest descent and conjugate gradient

```
  cout << "The Matrix A that we are using: " << endl;
  A.Print();
  cout << endl;
  x = ConjugateGradient(A,b,x0);
  xsd = SteepestDescent(A,b,x0);
  cout << "The approximate solution using Conjugate Gradient is: " << endl;
  x.Print();
  cout << endl;
  cout << "The approximate solution using Steepest Descent is: " << endl;
  xsd.Print();
  cout << endl;
```

Simple codes for steepest descent and conjugate gradient

```
Vector SteepestDescent(Matrix A, Vector b, Vector x0){
  int IterMax, i;
  int dim = x0.Dimension();
  const double tolerance = 1.0e-14;
  Vector x(dim),f(dim),z(dim);
  double c,alpha,d;
  IterMax = 30;
  x = x0;
  f = A*x-b;
  i = 0;
```

Simple codes for steepest descent and conjugate gradient

```
  while (i <= IterMax){
    z = A*f;
    c = dot(f,f);
    alpha = c/dot(f,z);
    x = x - alpha*f;
    f =  A*x-b;
    if(sqrt(dot(f,f)) < tolerance) break;
    i++;
  }
  return x;
}
```

Simple codes for steepest descent and conjugate gradient

```
Vector ConjugateGradient(Matrix A, Vector b, Vector x0){
int dim = x0.Dimension();
const double tolerance = 1.0e-14;
Vector x(dim),r(dim),v(dim),z(dim);
double c,t,d;

x = x0;
r = b - A*x;
v = r;
c = dot(r,r);
```

Simple codes for steepest descent and conjugate gradient

```
int i = 0; IterMax = dim;
while(i <= IterMax){
  z = A*v;
  t = c/dot(v,z);
  x = x + t*v;
  r = r - t*z;
  d = dot(r,r);
  if(sqrt(d) < tolerance)
```

```
      break;
    v = r + (d/c)*v;
    c = d;  i++;
  }
  return x;
}
```

Codes from numerical recipes

The codes are taken from chapter 10.7 of Numerical recipes. We use the functions *dfpmin* and *lnsrch*. You can load down the package of programs from the webpage of the course, see under project 1. The package is called *NRcgm*107*.tar.gz* and contains the files *dfmin.c*, *lnsrch.c*, *nrutil.c* and *nrutil.h*. These codes are written in C.

```
void dfpmin(double p[], int n, double gtol, int *iter, double *fret,
double(*func)(double []), void (*dfunc)(double [], double []))
```

What you have to provide

The input to **dfpmin**

```
void dfpmin(double p[], int n, double gtol, int *iter, double *fret,
double(*func)(double []), void (*dfunc)(double [], double []))
```

is

- The starting vector p of length n
- The function $func$ on which minimization is done
- The function $dfunc$ where the gradient i calculated
- The convergence requirement for zeroing the gradient $gtol$.

It returns in p the location of the minimum, the number of iterations and the minimum value of the function under study $fret$.

Simple example and code (model.cpp on webpage)

```
#include "nrutil.h"
using namespace std;
//     Here we define various functions called by the main program

double E_function(double *x);
void   dE_function(double *x, double *g);
void   dfpmin(double p[], int n, double gtol, int *iter, double *fret,
           double(*func)(double []), void (*dfunc)(double [], double []));
//   Main function begins here
int main()
{
     int n, iter;
     double gtol, fret;
```

```
     double alpha;
     n = 1;
     cout << "Read in guess for alpha" << endl;
     cin >> alpha;
```

Simple example and code (model.cpp on webpage)

```
//   reserve space in memory for vectors containing the variational
//   parameters
     double *p = new double [2];
     gtol = 1.0e-5;
//   now call dfmin and compute the minimum
     p[1] = alpha;
     dfpmin(p, n, gtol, &iter, &fret,&E_function,&dE_function);
     cout << "Value of energy minimum = " << fret << endl;
     cout << "Number of iterations = " << iter << endl;
     cout << "Value of alpha at minimu = " << p[1] << endl;
      delete [] p;
```

Simple example and code (model.cpp on webpage)

```
//  this function defines the Energy function
double E_function(double x[])
{
  double value = x[1]*x[1]*0.5+1.0/(8*x[1]*x[1]);
  return value;
} // end of function to evaluate
```

Simple example and code (model.cpp on webpage)

```
//  this function defines the derivative of the energy
void dE_function(double x[], double g[])
{
  g[1] = x[1]-1.0/(4*x[1]*x[1]*x[1]);
} // end of function to evaluate
```

Simple example and code (model.cpp on webpage)

```
using namespace std;
//     Here we define various functions called by the main program

double E_function(double *x);
void   dE_function(double *x, double *g);
void   dfpmin(double p[], int n, double gtol, int *iter, double *fret,
           double(*func)(double []), void (*dfunc)(double [], double []));
//   Main function begins here
int main()
{
     int n, iter;
```

```
     double gtol, fret;
     double alpha;
     n = 1;
     cout << "Read in guess for alpha" << endl;
     cin >> alpha;
```

Simple example and code (model.cpp on webpage)

```
//   reserve space in memory for vectors containing the variational
//   parameters
     double *p = new double [2];
     gtol = 1.0e-5;
//   now call dfmin and compute the minimum
     p[1] = alpha;
     dfpmin(p, n, gtol, &iter, &fret,&E_function,&dE_function);
     cout << "Value of energy minimum = " << fret << endl;
     cout << "Number of iterations = " << iter << endl;
     cout << "Value of alpha at minimu = " << p[1] << endl;
      delete [] p;
```

Simple example and code (model.cpp on webpage)

```
//  this function defines the Energy function
double E_function(double x[])
{

//  Change here by calling your Metropolis function which
//  returns the local energy

  double value = x[1]*x[1]*0.5+1.0/(8*x[1]*x[1]);
  return value;
} // end of function to evaluate
```

You need to change this function so that you call the local energy for your system. I used 1000 cycles per call to get a new value of $\langle E_L[\alpha]\rangle$.

Simple example and code (model.cpp on webpage)

```
//  this function defines the derivative of the energy
void dE_function(double x[], double g[])
{
//  Change here by calling your Metropolis function.
//  I compute both the local energy and its derivative for every call to func

  g[1] = x[1]-1.0/(4*x[1]*x[1]*x[1]);
} // end of function to evaluate
```

You need to change this function so that you call the local energy for your system. I used 1000 cycles per call to get a new value of $\langle E_L[\alpha]\rangle$. When I compute the local energy I also compute its derivative. After roughly 10-20 iterations I got a converged result in terms of α.