Cornell University

Department of Physics

Phys 480/680, Astro 690 April 15, 2005

Computational Physics, Spring 2005

Homework Assignment # 8

(Due Thursday, March 21)

Agenda and readings for the weeks of April 18-22:

Goals: Implement numerical minimization using preconditioning, steepest descents, conjugate gradient techniques; solve Kohn-Sham equations of density-functional theory.

Readings marked NR are from *Numerical Recipes: The Art of Scientific Computing*, 2nd edition (in C). Readings marked LN are from the course lecture notes to be found at http://www.ccmr.cornell.edu/~muchomas/P480.

Recap of last two weeks of class

- 04/05 Lec 18: Discrete Fourier Transforms: aliasing/Nyquist frequency, Fourier factorization theorem, Fourier inversion theorem
  Reading: NR 12.0, 12.1, 12.4
- 04/07 Lec 19: Eigenvalue problems in $d > 1$ dimension: variational principle, analytic continuation for constraints, minimization by steepest descents
  Reading: NR 10.0
- 04/07 Lab 6: FLOP performance of basic operations; FLOP rates in octave; Automated empirical optimization of software (AEOS): automatically tuned linear algebra software (ATLAS), fastest Fourier Transform in the West (FFTW).
- 04/12 Lec 20: DFT++ expressions for energy and gradient for multiple states and for density functional theory
  Reading: Sections 1–4 of http://arxiv.org/abs/cond-mat/9909130
- 04/14 Lec 21: Converge of steepest descents; preconditioning; line minimization; motivation for conjugate gradients
  Reading: NR 10.2, 10.3. 10.5
- 04/14 Lab 7: Spectral solution of Poisson's and Schrödinger's equations

Preview of next week of class:

- 04/19 Lec 22: Conjugate gradient algorithm; extraction of eigenstats from variational principle
  Reading: 10.6
- 04/21 Lec 23: FFT
  Reading: 12.2
- 04/21 Lab 8: Converge of steepest descents; preconditioning; line minimization; quantum states of harmonic oscillator, quantum dots

Contents

1 FFTW **3**

2 Advanced techniques for numerical minimization **4**
2.1 sd() . . . 4
2.2 lm() . . . 4
2.2.1 Debugging . . . 5
2.3 K() . . . 5
2.4 pclm() . . . 6
2.4.1 Debugging . . . 6
2.5 pccg() . . . 7
2.5.1 Debugging . . . 8
2.6 Final comparison of techniques . . . 8

3 Density functional theory: "dft.m" **8**
3.1 Background . . . 8
3.2 Implementation strategy: "dft.m" . . . 10
3.3 Occupancies . . . 10
3.3.1 Debugging . . . 10
3.4 Hartree theory . . . 10
3.4.1 Debugging . . . 10
3.5 Full density-functional theory . . . 11
3.5.1 Debugging . . . 11
3.6 Quantum dot . . . 11

A "excVWN.m" **12**

B "excpVWN.m" **13**

1 FFTW

The version of octave installed on the workstations of the Physics Education Cluster (the "ws##" machines, not the login node "ac") actually have FFTW installed.

To get an idea of the performance enhancements associated with this package, first time the current "fft3.m" package which you are using:

1. make sure you are on one of the "ws##" machines and not the login node "ac";
2. make sure you are in your current working directory with the "poisson.m" code;
3. edit "setup.m" so that you use a grid of size $64 \times 64 \times 64$;
4. edit "poisson.m" so that you comment out the plotting calls to mesh();
5. overwrite your current "fft3.m" with a version that includes an MFLOPS counter:

```
cp /home/muchomas/P480/Octave/fft3_factored.m fft3.m
```

6. run and time the Poisson calculation a few times within octave with

```
octave> setup
octave> tic; poisson; runtime=toc
octave> tic; poisson; runtime=toc
octave> tic; poisson; runtime=toc
```

The code will then print out the MFLOPS rate for each three-dimensional FFT call and also print out your final runtime. (I obtain a total runtime of about 0.73 seconds and a processing rate of about 30 MFLOPS.)

Next, install the direct call to the FFTW package with

```
cp /home/muchomas/P480/Octave/fft3w.m fft3.m
```

and rerun the above timing test with

```
octave> setup
octave> tic; poisson; runtime=toc
octave> tic; poisson; runtime=toc
octave> tic; poisson; runtime=toc
```

You should find that you get exactly the same results as before but at a notable improvement in performance.

Finally, before proceeding with the rest of this assignment, you may find it convenient to comment out the lines of code used in your final version of "fft3.m" to compute the MFLOPS rate, namely the two with the "tic" and "MFLOPS=..." commands.

2 Advanced techniques for numerical minimization

Before proceeding, to make sure that everyone works with the same problem size, be sure to set

```
S=[20; 25; 30];
```

in "setup.m".

In this problem, we implement preconditioning, line minimization, and conjugate gradients. To track the convergence properties of the various techniques, each function will return, as an optional second argument, the list of energies at each stage of the minimization. We begin by adding this capability to sd().

2.1 sd()

```
function [out, Elist]=sd(W,Nit)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired

Output:

- out: Result of Nit iterations of steepest descents
- Elist: Nit×1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE() also displayed

Generalize your function in "sd.m" to provide the above functionality. It practice, you can accomplish this by including "Elist" in the function prototype at the top of the file and adding the two indicated statements to your loop as in the example below.

```
% Minimization with steepest descent algorithm using getE() and getgrad()
%
% Usage: [out, Elist]=sd(W,Nit)

function [out, Elist]=sd(W,Nit)
          .
          .
          .
  for it=1:Nit
    out=out-alpha*getgrad(out);
    Elist(it)=getE(out); %# <= New statements
    Elist(it)
  end
          .
          .
          .
```

2.2 lm()

```
function [out, Elist]=lm(W,Nit)
```

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired

Output:

- out: Result of Nit iterations of line minimization
- Elist: Nit×1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE(), "linmin test" (angle cosine between previous search direction and current gradient)

Implement the line minimization algorithm discussed in lecture,

1. Initialize W_0
2. Evaluate gradient: $g = \nabla_W E(W_n)$
3. If not first iteration, do "linmin test": check $(g \cdot d)/\sqrt{(g \cdot g)(d \cdot d)}$ for d being the previous search direction
4. Set search direction: $d = -g$
5. Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d)$
6. Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - gt] \cdot d)$
7. $W_{n+1} = W_n + \alpha d$
8. Repeat (2-7)

with a function of the above prototype in the file "lm.m". You may wish to experiment with the trial step size. You will find that the algorithm is not very sensitive to this parameter. A trial step size of $\alpha_t = 3 \times 10^{-5}$ to work well.
Hint: As on the previous assignment, because of our trick of coding gradients as complex matrices, we must evaluate the dot products according to the formula

$$a \cdot b = 2\mathrm{Re}\,\mathrm{Tr}\,\left(a^\dagger b\right).$$

As a matter of convenience, note that the "extra" factor of two, while very important for getting the values of dot products correct, actually cancels out of all of the expressions above.

2.2.1 Debugging

Replace the call to sd() in your "sch.m" script file with lm(), and run. If you have implemented steepest descents correctly, you should find that the energy always decreases and that the angle cosine test gives nearly zero, indicating that the new gradient is orthogonal to the previous search direction. Note that at the start, when you are far from the minimum and the function is not very quadratic, the approximate minimization is not perfect and you may see angle cosines near 0.1. However, after about 20 iterations, the algorithm begins to approach the minimum, and the angle cosines will become much smaller ($\sim 10^{-4}$) and continue to improve as the minimization proceeds.

2.3 K()

`function out=K(W)`

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions

Global variables:

- gbl_G2: Square magnitude of G vectors

Output:

- out: Preconditioner 1/(1+G2) applied to each column of W

Implement in the file "K.m" the preconditioner function of the above prototype.
Hint: Your code should closely resemble that for L().

2.4 pclm()

`function [out, Elist]=pclm(W,Nit)`

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired

Output:

- out: Result of Nit iterations of preconditioned line minimization
- Elist: Nit×1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE(), "linmin test" (angle cosine between previous search direction and current gradient)

Implement the preconditioned line minimization algorithm discussed in lecture,

1. Initialize W_0
2. Evaluate gradient: $g = \nabla_W E(W_n)$
3. If not first iteration, do "linmin test": check $(g \cdot d)/\sqrt{(g \cdot g)(d \cdot d)}$ for d being the previous search direction
4. Set search direction: $d = -K(g)$
5. Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d)$
6. Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - gt] \cdot d)$
7. $W_{n+1} = W_n + \alpha d$
8. Repeat (2-7)

with a function of the above prototype in the file "pclm.m".
Hint: You should have to change only a single line of "lm.m" to generate "pclm.m"

2.4.1 Debugging

The ultimate test of a good preconditioner is whether is actually improves the rate of convergence. You may note, however, that if you replace your current call to lm() in "sch.m" with pclm(), the energy may begin to increase momentarily. This is not uncommon when starting a minimization far from the minimum. To make a more appropriate comparison replace your current call to lm() with

```
W=lm(W,20); %# 20 iterations of simple lm() to get nearer to the minimum
[Wlm, Elm]=lm(W,50); %# 50 iterations of lm() from W, while recording results
[Wpclm, Epclm]=pclm(W,50); %# 50 iterations of pclm from same W
semilogy([1:50],Elm-18,'r-',[1:50],Epclm-18,'b-'); %# Plot results on log scale
pause; %# Wait for keyboard input before continuing
```

This will first "relax" W with 20 iterations to give a joint starting point reasonably close to the minimum, run both lm() and pclm() from that same point for 50 iterations, and plot on a log scale the difference of each algorithm from the analytic result of 18, with line minimization in red, and preconditioned line minimization in blue. If the preconditioner is working you should notice *both* a large initial difference coming from the first iteration where the preconditoner eliminates nearly all of the high frequency errors *and* a difference in slope (exponential convergence rate) at the higher iteration numbers.
Note: After the plot, the program will halt at the "pause;" statement. Hitting CNTRL-C at this point will regain the octave prompt. Alternately, hitting any other key will let the program continue on to the plots of the wave functions stored in W.

2.5 pccg()

`function [out, Elist]=pccg(W,Nit,cgform)`

Input:

- W: $\prod S \times N_s$ matrix containing initial guess for eigensolutions
- Nit: Number of iterations desired
- cgform: Version of cg: 1=Fletcher-Reeves, 2=Polak-Ribiere, 3=Hestenes-Stiefel

Output:

- out: Result of Nit iterations of preconditioned conjugate gradients
- Elist: Nit×1 column vector containing result of getE() after each iteration
- DISPLAY: result of each getE(), "linmin test" (angle cosine between previous search direction and current gradient), "conjugate gradient test" (angle cosine, measured through the preconditioner, of the current gradient and the previous gradient)

Implement the preconditioned conjugate-gradients algorithm discussed in lecture,

1. Initialize W_0
2. Evaluate gradient: $g_n = \nabla_W E(W_n)$
3. If not first iteration, do "linmin test": check $(g_n \cdot d_{n-1})/\sqrt{(g_n \cdot g_n)(d_{n-1} \cdot d_{n-1})}$ for d being the previous search direction
4. If not first iteration, do "cg test": check $(g_n \cdot Kg_{n-1})/\sqrt{(g_n \cdot Kg_n)(g_{n-1} \cdot Kg_{n-1})}$ for d being the previous search direction
5. Set search direction: $d_n = -Kg_n + \beta d_{n-1}$, where
   (a) Fletcher-Reeves: $\beta = (g_n \cdot Kg_n)/(g_{n-1} \cdot Kg_{n-1})$
   (b) Polak-Ribiere: $\beta = ([g_n - g_{n-1}] \cdot Kg_n)/(g_{n-1} \cdot Kg_{n-1})$
   (c) Hestenes-Stiefel: $\beta = ([g_n - g_{n-1}] \cdot Kg_n)/([g_n - g_{n-1}] \cdot d_{n-1})$
6. Evaluate gradient at trial step: $g_t = \nabla_W E(W_n + \alpha_t d_n)$
7. Compute estimate of best step: $\alpha = \alpha_t (g \cdot d)/([g - g_t t] \cdot d_n)$
8. $W_{n+1} = W_n + \alpha d_n$
9. Repeat (2-8)

with a function of the above prototype in the file "pccg.m".
Hint: You may wish to use the "`if ... elseif ... elseif ... end`" octave contstruct.

2.5.1 Debugging

Analogously to debugging for line minimization, the best test of your implementation is the gradient-orthogonality theorem, the "cg test". Depending upon the form used for β, this test will not function well until quite close to the minimum. Thus, begin your testing with the least sensitive version (Hestenes-Stiefel), running it from the wave functions determined by preconditioned line minimization by entering "`Wout=pccg(Wpclm,50,3);`" at the command prompt after having just run your "sch.m". You should observe that "cg test" gives results which track but are somewhat larger than those of the "linmin test". Also, after a number of iterations, the "cg test" should become quite good.

Once this test passes, Wout will contain wave functions sufficiently close to the minimum that the other forms for β will work well when properly implemented. You may then verify those versions using "`pccg(Wout,10,1);`" and "`pccg(Wout,10,2);`".

2.6 Final comparison of techniques

To complete the comparison of the techniques, replace the line

```
semilogy([1:50],Elm-18,'r-',[1:50],Epclm-18,'b-'); %# Plot results on log scale
```

in your current version of "sch.m" with the code block

```
[Wcg1, Ecg1]=pccg(W,50,1); %# 50 iters from W of cg versions 1, 2, 3 ...
[Wcg2, Ecg2]=pccg(W,50,2);
[Wcg3, Ecg3]=pccg(W,50,3);
%# Plot results on log scale
its=[1:50];
semilogy(its,Elm-18,its,Epclm-18,its,Ecg1-18,its,Ecg2-18,its,Ecg3-18);
%# Set W to best result for plotting
W=Wcg3;
```

Be prepared to show your final plot to the grader.

3 Density functional theory: "dft.m"

3.1 Background

At this stage, the only differences between your Schrödinger solver and a density functional solver are the energy and gradient functions, getE(), getgrad() and getH(), respectively.

To see this, we begin by noting that one important difference between the energy functions is that, now, each wave function ψ_i has an associated occupancy, or "filling factor," f_i. For simplicity we shall here always work in cases where all states have the same occupancy $f_i = f$. (Many physical systems have this property; usually $f = 2$. The generalization to difference occupancies for different orbitals is conceptually straight forward but involves a *lot* of messy algebra.)

Under the conditions of the previous paragraph, three of the quantities from density-functional theory have very similar forms to those from the Schrödinger case. For instance, the density-functional form for the kinetic energy $(-1/2)\sum_i f_i \int \psi_i^* \nabla^2 \psi_i$ simply picks up an extra factor of f,

$$KE = -\frac{1}{2}\sum_i f_i \int \psi_i^* \nabla^2 \psi_i = f \cdot \frac{1}{2}\mathrm{Tr}\,(W^\dagger L W N^{-1/2}). \tag{1}$$

The total electron density $\sum_i f_i |\psi_i|^2$ picks up a similar factor of f,

$$\vec{n} = f \cdot \mathrm{diag}\left[\mathcal{I}WN^{-1}\left(\mathcal{I}W\right)^2\right]. \tag{2}$$

Finally, in terms of the vector of sample values $\vec{n}$, the formula for the electron-nuclear potential energy $PE = \int V(r)n(r)$ is identical to the corresponding formula in the Schrödinger case and thus leads to the same result,

$$PE = \tilde{V}^\dagger \vec{n}, \tag{3}$$

where

$$\tilde{V} \equiv \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \vec{V}, \tag{4}$$

with $\vec{V}$ being the sample values of the potential.

The first new term appearing in density-functional theory is the electron-electron potential energy $U_{ee} = \frac{1}{2}\int n(r)^* \phi(r)$, where $\nabla^2 \phi(r) = -4\pi n(r)$. (As in a similar derivation in lecture, the "*" on $n(r)$ doesn't change the integral because $n(r)$ is real. We include it, however, so that we may use our currently defined operator $\mathcal{O}$.) Inserting the expansions $n(r) = \sum_\alpha b_\alpha(r)\hat{n}_\alpha$ and $\phi(r) = \sum_\alpha b_\alpha(r)\hat{\phi}_\alpha$ into the integral for U_{ee}, using our solution for Poisson's equation from Problem Set 7 and rearranging quantities within †'s, we find

$$U_{ee} = \frac{1}{2}\vec{n}^\dagger \mathcal{J}^\dagger \mathcal{O} \hat{\phi}, \tag{5}$$

where the expansion coefficients for the solution to Poisson's equation are

$$\hat{\phi} = -4\pi L^{-1} \mathcal{O} \mathcal{J} \vec{n}.$$

The final term we require is the exchange correlation energy $\int n_(r)^* \epsilon_{xc}(n(r))$. Defining the operator $\epsilon_{xc}(\vec{n})$ as returning a vector each of whose components is the evaluation of the exchange-correlation energy for the corresponding component of $\vec{n}$ and inserting appropriate expansions gives the result,

$$E_{xc} = \vec{n}^\dagger \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \epsilon_{xc}(\mathcal{I}\vec{n}). \tag{6}$$

In summary, in terms of unconstrained wave function coefficients W, the total energy within density functional theory is

$$\begin{aligned} E_{LDA} &= -f \cdot \frac{1}{2}\mathrm{Tr}\left(W^\dagger L W N^{-1}\right) + \tilde{V}^\dagger n \\ &\quad + \frac{1}{2}\vec{n}^\dagger \mathcal{J}^\dagger \mathcal{O} \hat{\phi} + \vec{n}^\dagger \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \epsilon_{xc}(\vec{n}), \end{aligned} \tag{7}$$

where

$$\begin{aligned} N &\equiv W^\dagger \mathcal{O} W \\ \vec{n} &\equiv f \cdot \mathrm{diag}\left[\mathcal{I} W N^{-1} \left(\mathcal{I} W\right)^\dagger\right] \\ \hat{\phi} &\equiv -4\pi L^{-1} \mathcal{O} \mathcal{J} \vec{n}. \end{aligned}$$

As with the Schrödinger case, two effects contribute to the gradient. First, there are contributions due to the constraints, which turn out to be of an identical form as we had previously. Including the appropriate factors for f, we thus have

$$\nabla_W E = f\left(HW - \mathcal{O}WN^{-1}W^\dagger HW\right)N^{-1}. \tag{8}$$

Finally, the operator $H()$ encapsulates the remaining contributions to the gradient, which come from the basic structure of the energy and has the form,

$$H \equiv -\frac{1}{2}L + \mathcal{I}^\dagger \left(\mathrm{Diag}\, V_{eff}\right)\mathcal{I}, \tag{9}$$

where

$$V_{eff} \equiv \tilde{V} + \mathcal{J}^\dagger \mathcal{O} \hat{\phi} + \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \epsilon_{xc}(\vec{n}) + \mathrm{Diag}\left[\epsilon'_{xc}(\vec{n})\right] \mathcal{J}^\dagger \mathcal{O} \mathcal{J} \vec{n},$$

with the operator $\epsilon'_{xc}(\vec{n})$ defined similarly to $\epsilon_{xc}(\vec{n})$ but now evaluating $\partial \epsilon_{xc}/\partial n$ on each component of $\vec{n}$.

3.2 Implementation strategy: "dft.m"

In principle, we need now only modify getE(), getgrad() and H() to perform density functional calculations. In order not to destroy your Schrödinger solver, copy each of the ".m" files and "fftw3.oct" in your current directory to a new directory where we shall now work. Once this is done, go to your new directory, rename "sch.m" to "dft.m", and run "dft.m" from within octave to confirm that it behaves just as does "sch.m". (Don't forget to run "setup.m" first!)

Next, we shall make the appropriate changes to getE(), getgrad() and getH(). Because the minimization algorithms are already debugged, we need first only confirm we have an appropriate energy and gradient with the finite difference test, and then we are ready for full density-functional calculations!

To prevent proliferation of output as we run these tests, you may wish to put a "`pause;`" statement immediately after the call to fdtest() in "dft.m". Again, you should run "dft.m" one last time to confirm all is well.

3.3 Occupancies

The various energy and gradient functions require access to the filling filling factor f. As this is the final assignment, rather than passing f as an argument down through all of the various functions, we will allow ourselves one final global variable, "gbl_f". Add the following lines to the very top of "dft.m",

```
%# Set the orbital occupancies
global gbl_f;
gbl_f=2; %# The usual case of a constant filling of two electrons per orbital
```

Your original Schrödinger solver implicitly assumed state occupancies of $f = 1$. The above energy and gradient formulas for density-functional theory should work just as well for general values of f, even if the ϕ and ϵ_{xc} terms are set to zero. Thus, before adding any of the new terms, add the factors of f to the appropriate places in getE() and getgrad().

Hints: Don't forget to declare gbl_f as a global variable in each function which needs "f" and to include it in both the density and the kinetic energy parts.

3.3.1 Debugging

Run "dft.m" and verify proper functioning of the finite-difference test.

3.4 Hartree theory

Ignoring the exchange-correlation terms represents the first mean-field many-electron theory, "Hartree theory." We will begin with this theory as it involves energy terms with which we are already familiar.

Add all terms to getE() and H() from Eqs. (7,9) which involve ϕ.

Hint: Note that you will have to build the density n and solve Poisson's both in get() and in H().

3.4.1 Debugging

Because of possible cancellations, the most convincing test is to first zero out all terms except the Hartree terms in the energy and gradient before running the finite difference test. (Any easy way to do this is without much typing is to just multiply the unwanted terms temporarily by zero with "`*0`".) Once you have confirmed your Hartree terms, then turn all of the other terms back on and run the test again to make sure everything is turned on properly.

3.5 Full density-functional theory

Add the remaining terms in Eqs. (7,9), those involving ϵ_{xc} and ϵ'_{xc}, to get() and H(). You may use the functions excVWN() and excpVWN() in Appendices A and B, which implement the operators $\epsilon_{xc}(\vec{n})$ and $\epsilon'_{xc}(\vec{n})$, respectively, from the same functions used in our $d = 1$ dimensional solver for atoms.

3.5.1 Debugging

Run the finite difference test, first zeroing out all terms but those involving ϵ_{xc} and then including all terms.

3.6 Quantum dot

A common approximation for a "quantum dot," a relatively small number of electrons trapped in a potential, is to model the trapping potential as a harmonic oscillator (which actually is the potential V(r) which you have currently in "dft.m") and to treat the electron-electron interactions within density-functional theory. To perform such a calculation, replace the code block in "dft.m",

```
%# Converge using steepest descents
W=lm(W,20); %# 20 iterations of simple lm() to get close to the minimum
[Wlm, Elm]=lm(W,50); %# 50 iterations of lm() from W, while recording results
                .
                .
                .
semilogy(its,Elm-18,its,Epclm-18,its,Ecg1-18,its,Ecg2-18,its,Ecg3-18);
%# Set W to best result for plotting
W=Wcg3;
pause;
```

which compares convergence of various minimization methods (now inappropriately assuming that the converged result is 18) with

```
%# Be sure to show lots of digits!
format long

%# Converge a little using steepest descents
W=sd(W,10);

%# Converge more with preconditioned line minimization
W=pclm(W,20);

%# Finish convergence with full preconditioned conjugate gradients
W=pccg(W,50,3);
```

Be sure to remove any "`pause;`" statements so that the code will also run through the part which gets the final wave functions and plots them.

Run your code. You should find a converged value of the total energy for this quantum dot of 8 electrons (4 states, with two electrons each) near Etot=43.337. You should also find states with similar s and p characters as to what you had before, but now with eigenvalues of approximately 5.509, 6.949, 6.949, 6.949.

A "excVWN.m"

```
% VWN parameterization of the exchange correlation energy
function out=excVWN(n)
  %# Constants
  X1 = 0.75*(3.0/(2.0*pi))^(2.0/3.0);
  A  =  0.0310907;
  x0 = -0.10498;
  b  = 3.72744;
  c  = 12.9352;
  Q  = sqrt(4*c-b*b);
  X0 = x0*x0+b*x0+c;

  rs=(4*pi/3*n).^(-1/3); %# Added internal conversion to rs

  x=sqrt(rs); X=x.*x+b*x+c;

  out=-X1./rs ...
    + A*( ...
        +log(x.*x./X)+2*b/Q*atan(Q./(2*x+b)) ...
        -(b*x0)/X0*( ...
                    log((x-x0).*(x-x0)./X)+2*(2*x0+b)/Q*atan(Q./(2*x+b)) ...
                    ) ...
        );
endfunction
```

B "excpVWN.m"

```
% d/dn deriv of VWN parameterization of the exchange correlation energy
function out=excpVWN(n)
  %# Constants
  X1 = 0.75*(3.0/(2.0*pi))^(2.0/3.0);
  A  =  0.0310907;
  x0 = -0.10498;
  b  = 3.72744;
  c  = 12.9352;
  Q  = sqrt(4*c-b*b);
  X0 = x0*x0+b*x0+c;

  rs=(4*pi/3*n).^(-1/3); %# Added internal conversion to rs

  x=sqrt(rs); X=x.*x+b*x+c;


  dx=(0.5)./x; %# Chain rule needs dx/drho!

  out=dx.*( ...
             2*X1./(rs.*x)+A*( ...
                             (2)./x-(2*x+b)./X-4*b./(Q*Q+(2*x+b).*(2*x+b)) ...
                             -(b*x0)/X0*((2)./(x-x0)-(2*x+b)./X-4*(2*x0+b)./ ...
                                                      (Q*Q+(2*x+b).*(2*x+b)) ) ..$.
                             ) ...
                             );

  out=(-rs./(3*n)).*out; %# Added d(rs)/dn from chain rule from rs to n conv

endfunction
```