A Crash Course in Numerical Analysis for Time-Dependent Density Functional Theory

Andrew Baczewski (Sandia)
Michele Pavanello (Rutgers)

TDDFT Summer School and Workshop
August 4-8, 2019

Sandia National Laboratories

CCR Center for Computing Research

Overview

Part 1: Numerical methods and standard DFT

Part 2: TDDFT and beyond

Problem statement

Consider the time-dependent Kohn-Sham equation:

$$i\frac{\partial}{\partial t}\phi_n(\mathbf{r},t) = \left(-\frac{\nabla^2}{2} + v_S\left[\rho\right](\mathbf{r},t)\right)\phi_n(\mathbf{r},t), \quad \forall \mathbf{r} \in \mathbb{R}^d,\ t \in [0,T]$$

$$\phi_n(\mathbf{r},t=0) = \phi_{n,0}(\mathbf{r}), \quad \rho_0(\mathbf{r}) = \sum_n f_n(T_e)|\phi_{n,0}(\mathbf{r})|^2, \quad \rho(\mathbf{r},t) = \sum_n f_n(T_e)|\phi_n(\mathbf{r},t)|^2$$

Initial condition on the KS orbitals

Closure between the density and orbitals

This is a system of nonlinear partial differential equations!

Often, numerical solutions are the best that we can do...

Elements of a numerical solution

What do we need to solve this problem?

$$i\frac{\partial}{\partial t}\phi_n(\mathbf{r},t) = \left(-\frac{\nabla^2}{2} + v_S\left[\rho\right](\mathbf{r},t)\right)\phi_n(\mathbf{r},t), \quad \forall\mathbf{r}\in\mathbb{R}^d,\ t\in[0,T]$$

$$\phi_n(\mathbf{r},t=0) = \phi_{n,0}(\mathbf{r}), \quad \rho_0(\mathbf{r}) = \sum_n f_n(T_e)|\phi_{n,0}(\mathbf{r})|^2, \quad \rho(\mathbf{r},t) = \sum_n f_n(T_e)|\phi_n(\mathbf{r},t)|^2$$

1.) Representation of the KS orbitals (+density+potential) at all points in time/space

(i.e., a choice of basis)

2.) Projection of the TDKS equations onto a finite-dimensional space

(i.e., a choice of discretization)

3.) Linearization of the nonlinear problem

(i.e., a choice of propagator)

4.) Solution of the finite-dimensional linear problem

(i.e., a choice of solver)

Numerical apologia

Before we go any further...

There is a deep and rigorous literature on numerical analysis

(i.e., it isn't all just finite differences)

If you're interested, we can try to point you in the right direction...

...but physicists/chemists/materials scientists often fall short of this standard by epsilon

(and that is OK!)

Choice of basis

Finite differences/real-space grids are an easy starting point for testing ideas
(of course, there are nice production codes that use these, too!)

Most CPU cycles go to codes that use either plane waves or Gaussians...
This popularity is almost entirely a function of numerical tricks!

Plane waves have uniform resolution - so we use pseudization to remove sharp features

Gaussians have adaptive resolution - but are overcomplete/may be ill-conditioned

There exist well-conditioned multi-resolution basis sets
(e.g., numerical atomic orbitals, wavelets, FEM-inspired)
but plane waves and Gaussians are surprisingly difficult to beat

Plane wave vs. Gaussians

$$|\psi_n\rangle = \frac{1}{\sqrt{\Omega}} \sum_{\mathbf{G}} c_n(\mathbf{G}) e^{i\mathbf{G}\cdot\mathbf{r}} |\mathbf{r}\rangle$$

Plane waves

- Closure with a caveat (i.e., uniform resolution)
- Coulomb kernel can be written as a spectral integral

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|} = 4\pi \int d\mathbf{q} \frac{e^{i\mathbf{q}\cdot(\mathbf{r}-\mathbf{r}')}}{|\mathbf{q}|^2}$$

- Screened exchange (a la HSE) was a huge advance
- Atom-centered grids used in pseudization (NCPP vs. USPP vs. PAW)
- Auxiliary basis sets are straightforward
- Fast Fourier transforms allow for fast matrix-vector multiplication

$$|\psi_n\rangle = \sum_{\alpha,\mathbf{R},l,m} c_n(\alpha,\mathbf{R}) e^{-\alpha|\mathbf{r}-\mathbf{R}|^2} Y_{lm}(\widehat{\mathbf{r}-\mathbf{R}})$$

Gaussians

- Closure, i.e., product of N Gaussians = 1 Gaussian
- Coulomb kernel can be written as a Gaussian integral

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|} = \frac{1}{\pi^{1/2}} \int_{-\infty}^{\infty} ds\ e^{-s^2|\mathbf{r}-\mathbf{r}'|^2}$$

- Great for Hartree-Fock + (post-HF)
- DFT required the implementation of grids into QC codes
- Resolution-of-identity tricks w/auxiliary basis sets
- Difficult to construct tricks for fast matrix-vector multiplication (though possible!)

Choice of discretization

*Almost** every method in electronic structure uses Galerkin discretization

(matrices are manifestly Hermitian)

$$\langle \phi | \hat{H}_{KS}[\rho] - \varepsilon | \psi_n \rangle = 0, \ \forall \phi \in T \subset L_2$$

For a fixed density, this reduces to an eigenproblem

Of course, we must seek a self-consistent solution

(NP-Complete** problem with good heuristics)

Optimal eigensolver depends on

1.) basis set efficiency, 2.) whether fast matrix-vector multiplication is available

Practically speaking, diagonalization-based approaches have a cubic scaling cost

*Finite difference methods are the most conspicuous exception...

**Whitfield, Schuch, and Verstraete, arXiv:1306.1259

Sub-cubic scaling methods

Methods that avoid diagonalization can achieve sub-cubic scaling!

$$E = \mathrm{Tr}\left[\hat{\rho}\hat{H}_{KS}\left[\rho\right]\right]$$

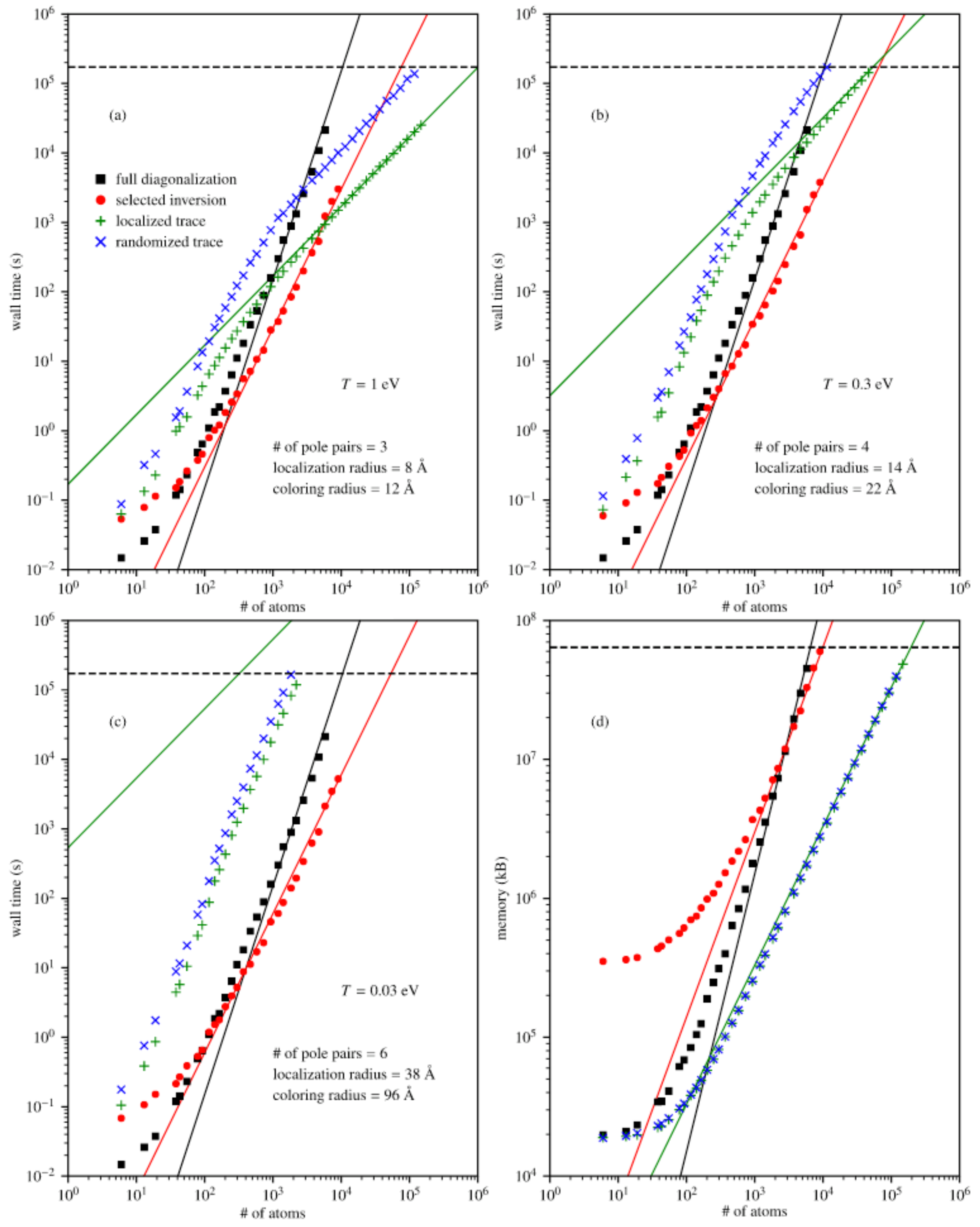
$$\hat{\rho} = \left[1 + e^{(\hat{H}_{KS}[\rho]-\mu\hat{N})/T}\right]^{-1}$$

Approximate the density matrix as:
1.) A polynomial in H
2.) A pole expansion (rational function of H)

Efficiently evaluate the trace:
1.) Exactly with selected inversion (Lin *et al.*, TOMS, 2011)
2.) Approximately with stochastic methods
3.) Approximately with forced sparsity
4.) Approximately with both 2 and 3

Constant prefactors are quite significant

Crossover between cubic and non-cubic scaling has remained at relatively large systems sizes for a long time



Moussa and Baczewski, Electronic Structure, 2019

Overview

Part 1: Numerical methods and standard DFT

Part 2: TDDFT and beyond

Problem statement (reminder)

Consider the time-dependent Kohn-Sham equation:

$$i\frac{\partial}{\partial t}\phi_n(\mathbf{r},t) = \left(-\frac{\nabla^2}{2} + v_S\left[\rho\right](\mathbf{r},t)\right)\phi_n(\mathbf{r},t), \quad \forall \mathbf{r} \in \mathbb{R}^d,\ t \in [0,T]$$

$$\phi_n(\mathbf{r},t=0) = \phi_{n,0}(\mathbf{r}), \quad \rho_0(\mathbf{r}) = \sum_n f_n(T_e)|\phi_{n,0}(\mathbf{r})|^2, \quad \rho(\mathbf{r},t) = \sum_n f_n(T_e)|\phi_n(\mathbf{r},t)|^2$$

Initial condition on the KS orbitals

Closure between the density and orbitals

This is a system of nonlinear partial differential equations!

Often, numerical solutions are the best that we can do...

Choice of propagator

$$i\frac{\partial}{\partial t}\phi_n(\mathbf{r},t) = \left(-\frac{\nabla^2}{2} + v_S\left[\rho\right](\mathbf{r},t)\right)\phi_n(\mathbf{r},t) = \hat{H}_{KS}\left[\rho\right]\phi_n(\mathbf{r},t)$$

Solution is easy to write down: $\phi_n(\mathbf{r},t) = \hat{U}(t,0)\phi_n(\mathbf{r},0)$ where $\hat{U}(t,0) = \mathcal{T}\left[e^{-i\int_0^t d\tau \hat{H}_{KS}[\rho(\tau)]}\right]$

Challenging to compute, in practice...

Approximate the time-ordered unitary as the composition of a bunch of small steps

Considerations include:

1.) Implicit vs. explicit scheme + self-consistency of density

2.) Order of convergence and stability

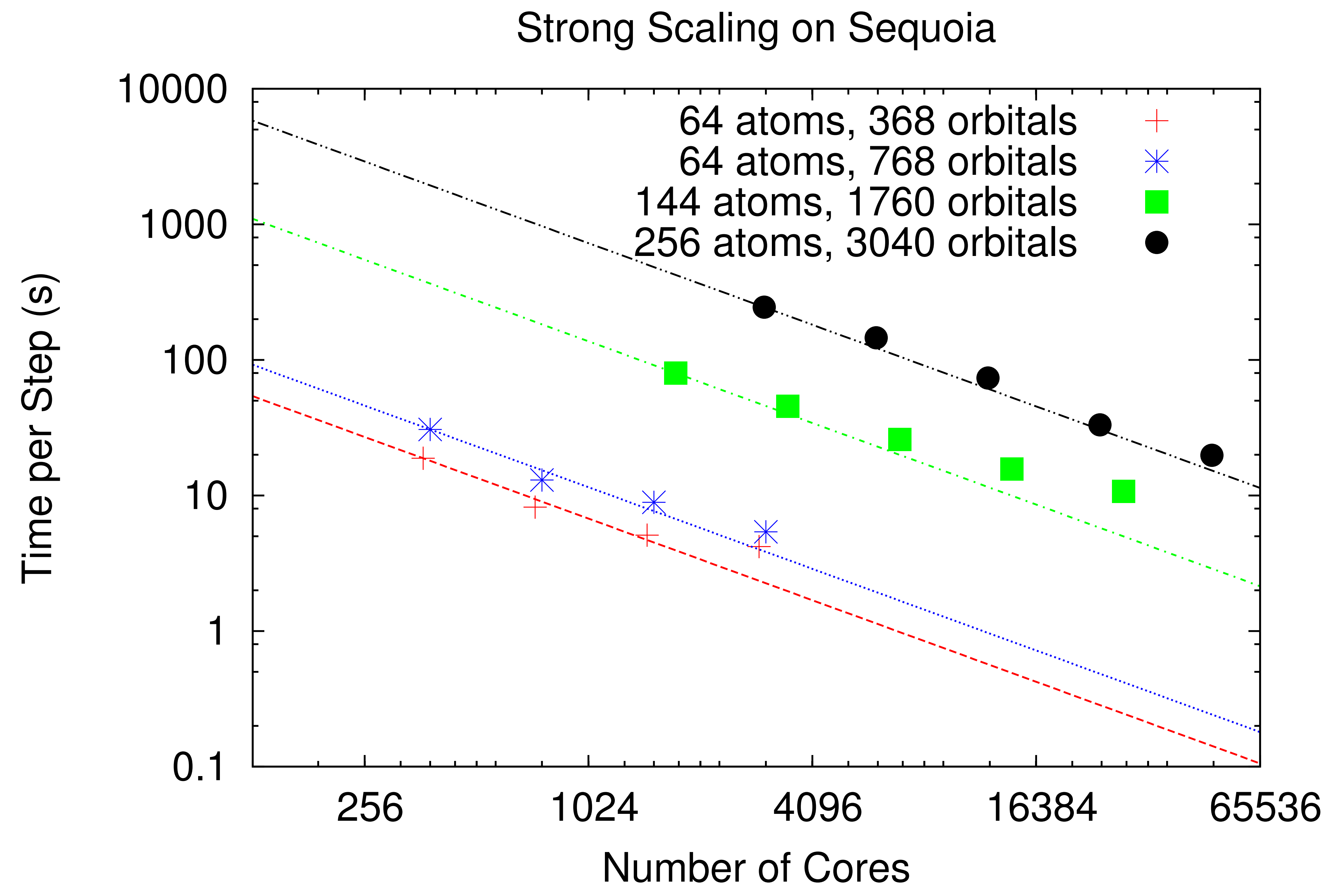
3.) Unitarity/symplecticity

4.) Whether nuclear motion is included

5.) Tricks for implementing matrix-vector multiplication

RT-TDDFT implementation details

- Many propagators only need simple matrix-vector multiplication - easy linear scaling!
- Some implementations rely on diagonalization at each time step
- Big difference between NCPP and PAW pseudization, especially for Ehrenfest dynamics
  - Inversion of overlap matrix
  - PAW + Ehrenfest leads to non-Hermitian effective Hamiltonian
  - Forces are a challenge to do “correctly”



Relatively easy to scale onto huge machines, embarrassingly parallel out to 1 task / orbital

LR-TDDFT

Unlike RT-TDDFT, this is a dense eigenproblem

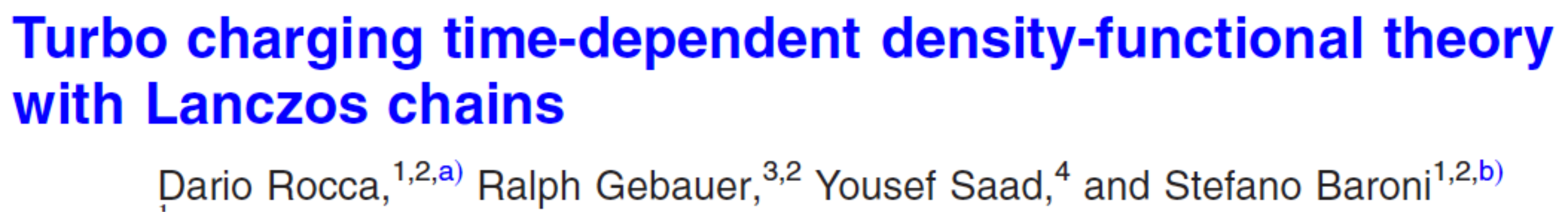
$$H = \begin{bmatrix} D_0 + \Phi^T f_{Hxc}\Phi & \Phi^T f_{Hxc}\Phi \\ -\Phi^T f_{Hxc}\Phi & -D_0 - \Phi^T f_{Hxc}\Phi \end{bmatrix}$$

Block dimension is $N_{occ}N_{virt} \times N_{occ}N_{virt}$

$$\Phi(\mathbf{r}) = [\phi_j(\mathbf{r})\phi_a^*(\mathbf{r})]\,,\ D_0 = \mathrm{diag}(\varepsilon_a - \varepsilon_j),\ \forall \text{ occupied } j,\ \text{virtual } a$$

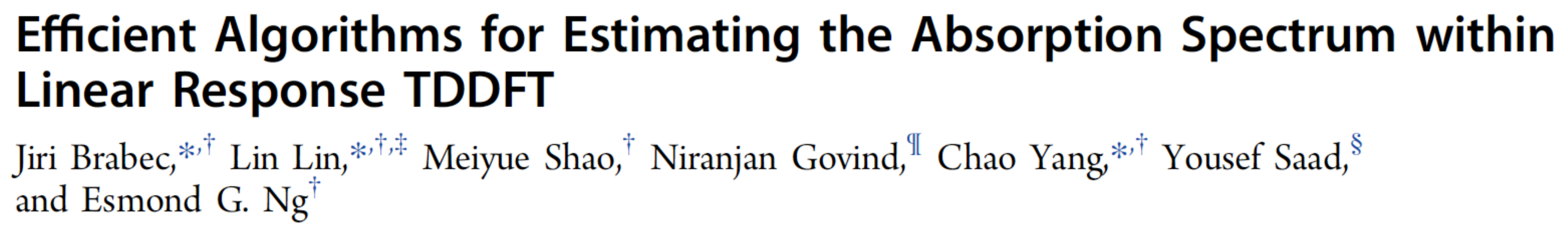
Approaches that reduce the cost:

1.) RT-TDDFT in LR "mode"
2.) Lanczos chains
3.) Kernel polynomial method



Turbo charging time-dependent density-functional theory with Lanczos chains

Dario Rocca,[1,2,a)] Ralph Gebauer,[3,2] Yousef Saad,[4] and Stefano Baroni[1,2,b)]



Efficient Algorithms for Estimating the Absorption Spectrum within Linear Response TDDFT

Jiri Brabec,[*,†] Lin Lin,[*,†,‡] Meiyue Shao,[†] Niranjan Govind,[¶] Chao Yang,[*,†] Yousef Saad,[§] and Esmond G. Ng[†]

Conclusion

You have now been exposed to the basics, what's next?

1.) Python notebooks are a nice start,
efficient/scalable implementations on HPC requires much more effort.

2.) If you want to get into code/method development,
being a good debugger = more time for science!

3.) It can be a real challenge to balance developing as a programmer with developing as a physicist/chemist/materials scientist. Talk to your mentors!

Happy hacking!

In [141]:
```
%matplotlib inline
%pylab inline
import scipy.sparse as sps
import scipy.sparse.linalg as spsl
```

```
Populating the interactive namespace from numpy and matplotlib
```

2019 TDDFT Summer School Hackathon

M. Pavanello (Rutgers) and A.D. Baczewski (Sandia)

Analytically sovable one-dimensional, one-electron problem

From Carsten Ullrich, https://arxiv.org/pdf/1305.1388.pdf (https://arxiv.org/pdf/1305.1388.pdf)

You are given the density of a one-electron system, $n(x)$, and are interested in solving for the potential and wave function from whence it came.

More precisely, the one-electron Schrödinger Equation in atomic units is,

$$\left[-\frac{1}{2}\frac{d^2}{dx^2} + v_s(x)\right]\phi(x) = \varepsilon\phi(x).$$

We are used to solving the problem in which we are given $v_s(x)$ - the functional form and/or a recipe for computing it from $\phi(x)$ - and then solving for $\phi(x)$ and ε. This typically takes for the form of an eigenproblem. Instead, the problem of determining $v_s(x)$ given a value of $n(x)$ becomes a simple algebraic one.

The particular choice of $n(x)$ that we will explore is the one studied in the above arXiv posting from Carsten,

$$n(x) = \cos^2(\pi x) + \cos^2(3\pi x), \quad x \in [-0.5, 0.5].$$

Be sure to note the domain of this function, for which the chosen functional form goes to zero at the boundaries. Of course, you should feel free to try different functional forms and/or domains once you're comfortable with this one.

Exercise 1

Define a uniform discretization of one-dimensional space and plot $n(x)$. Ask yourself,

1. Based on the functional form and the domain, how many points would you assume that you will need to resolve the salient features of the density by eye? For uniform / grid-based discretizations, if you can't resolve the features, neither can the computer.
2. Where does the density go to zero? What about first and second derivatives?
3. (Bonus) How would you define a non-uniform discretization? In realistic systems we can have species with very different Z values, large regions of vacuum (e.g., slabs/surfaces), or even regions with different densities (e.g., solid-liquid interfacesS). *(Note: Andrew can try to come up with a non-uniform density)*

Solution 1

We have adopted a variable naming convention that can be effective for students who are more familiar with, e.g., Fortran, than Python. That is, in spite of Python's "duck" typing, we are labeling the variables with a prefix according to a particular convention,

- b = Boolean (or logical, true/false)
- i = integer
- r = real
- z = complex
- ia = integer array (ra = real array, za = complex array)
- n = cardinal variables (e.g., the number of something)

In addition to clear naming conventions, we have also included built-in tests to verify the code in our solutions. This is a nice habit to get into, particularly as you write larger and more complicated codes.

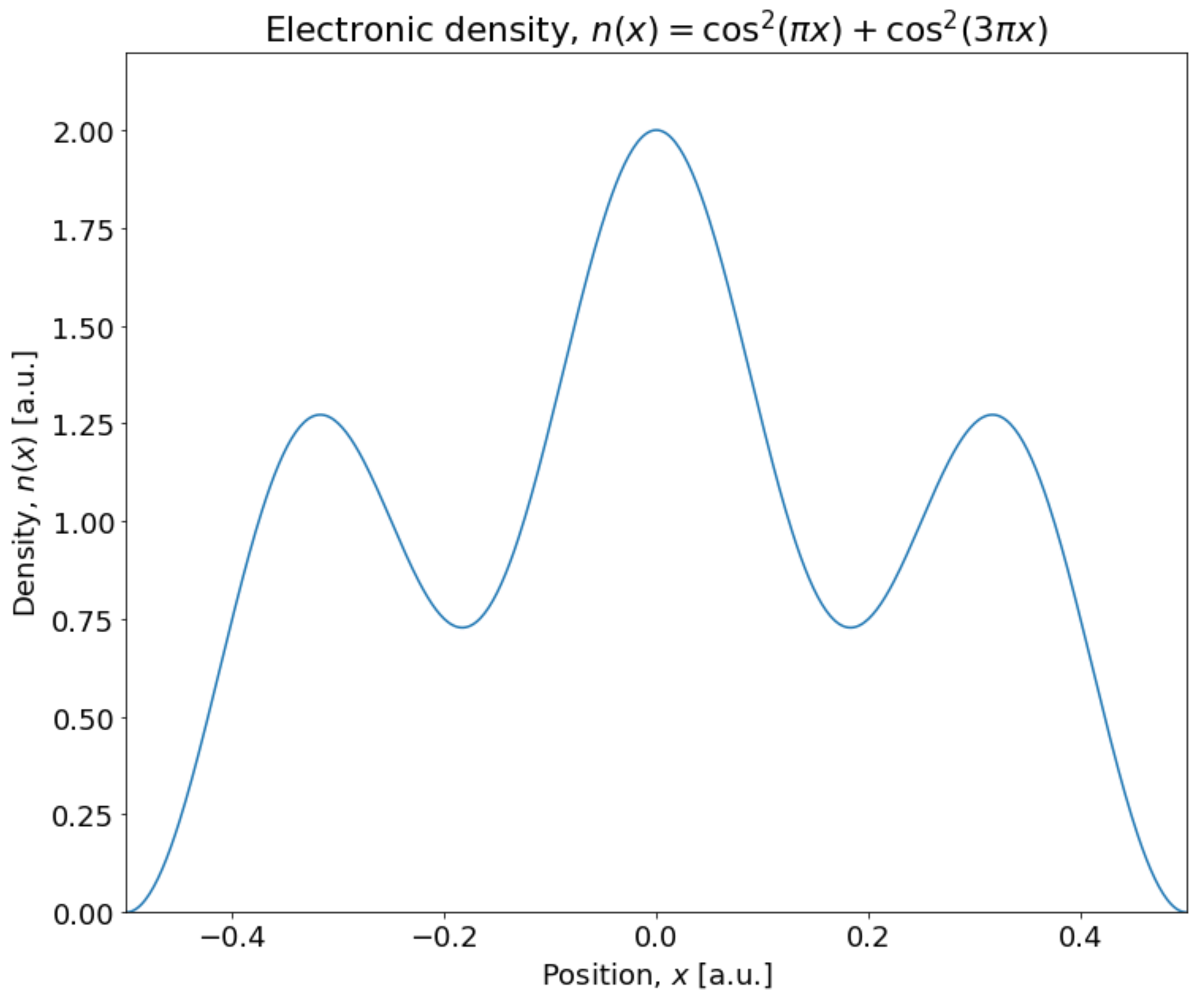
In [142]:
```
def getCarstenDensity( raX, bTestIntegral=False ):

    # implements the functional form from Carsten's arXiv posting
    raCarstenDensity = np.cos( np.pi * raX )**2 + np.cos( 3.0*np.pi * ra
X )**2

    # built-in test checks that the density is normalized, i.e., that it
corresponds to one electron
    if( bTestIntegral == True ):
        rDX = raX[1]-raX[0]
        rNorm = np.sum( raCarstenDensity ) * rDX
        print('[TEST] Integral of density computed using uniform quadrat
ure: ', rNorm)

    return raCarstenDensity
```

In [143]:
```
# start by trying 501 points, odd number is important for symmetry in va
lidation quadratures
# bonus question: why?
nPoints = 501

# specify the left and right end points of the domain
rLeft = -0.5
rRight = 0.5

# use the canned numpy routine for generating a uniformly spaced grid be
tween the two end points
raX = np.linspace( rLeft, rRight, nPoints )

# compute the density on a uniform grid
raCarstenDensity = getCarstenDensity( raX, bTestIntegral=True )
```

```
[TEST] Integral of density computed using uniform quadrature:  1.000000
0000000009
```

In [144]:
```
# plot the density
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
plot( raX, raCarstenDensity )

xlabel('Position, $x$ [a.u.]')
ylabel('Density, $n(x)$ [a.u.]')
title('Electronic density, $n(x) = \cos^2(\pi x) + \cos^2(3\pi x)$')
xlim([rLeft,rRight])
ylim([0.0,1.1*max( raCarstenDensity )])
```

Out[144]:
```
(0.0, 2.2)
```



Exercise 2

Prove that the potential, $v_s(x)$, must be

$$v_s(x) = \frac{\left(\frac{d^2 n(x)}{dx^2}\right)}{4n(x)} - \frac{\left(\frac{dn(x)}{dx}\right)^2}{8n(x)^2}.$$

Solution 2

Recall that

$$\left[-\frac{1}{2}\frac{d^2}{dx^2}+v_s(x)\right]\phi(x)=\varepsilon\phi(x).$$

For a single electron, $n(x)=|\phi(x)|^2$, so we can write $\phi(x)=\sqrt{n(x)}$. Here we have assumed that the wave function is real.

Plugging this into the Schrödinger equation,

$$\left[-\frac{1}{2}\frac{d^2}{dx^2}+v_s(x)\right]\sqrt{n(x)}=\varepsilon\sqrt{n(x)},$$

we are left with evaluating $\frac{d^2}{dx^2}\sqrt{n(x)}$.

The first derivative is given as,

$$\frac{d}{dx}\sqrt{n(x)}=\frac{\frac{dn}{dx}}{2(n(x))^{1/2}},$$

and the second derivative is given as,

$$\frac{d^2}{dx^2}\sqrt{n(x)}=\frac{\frac{d^2n}{dx^2}}{2(n(x))^{1/2}}-\frac{\left(\frac{dn}{dx}\right)^2}{4(n(x))^{3/2}}.$$

It is evident that we can simplify this by factoring out $\sqrt{n(x)}$,

$$\frac{d^2}{dx^2}\sqrt{n(x)}=\left[\frac{\frac{d^2n}{dx^2}}{2n(x)}-\frac{\left(\frac{dn}{dx}\right)^2}{4n(x)^2}\right]\sqrt{n(x)}.$$

Plugging this expression into the Schrödinger equation,

$$\left[-\frac{\frac{d^2n}{dx^2}}{4n(x)}+\frac{\left(\frac{dn}{dx}\right)^2}{8n(x)^2}+v_s(x)\right]\sqrt{n(x)}=\varepsilon\sqrt{n(x)},$$

which implies that it must be the case that

$$v_s(x)=C+\frac{\frac{d^2n}{dx^2}}{4n(x)}-\frac{\left(\frac{dn}{dx}\right)^2}{8n(x)^2},$$

where C is an arbitrary real constant that determines ε. Of course, the zero of energy is arbitrary and we can choose this to be whatever we want (i.e., $C=0$ is perfectly valid).

Exercise 3

Evaluate the derivative and second derivative of the density on your grid and use it to evaluate $v_s(x)$. Use analytical formulas or numerics (for a bonus). Ask yourself,

1. Do the results have zeros that match up with where you expect them?
2. Based on the formula for $v_s(x)$, where do you expect that it will be most sensitive to numerical pathologies? What do you have to do to sensibly evaluate $v_s(x)$?
3. (Bonus) How can I compute the first and second derivatives if the density is specified numerically? Code up a finite difference stencil, or some other mechanism for computing derivatives, that will compute the first and second derivatives for an arbitrary density that is specified numerically.

Solution 3

The expressions for the derivatives can be evaluated at the cost of recalling and applying elementary calculus.

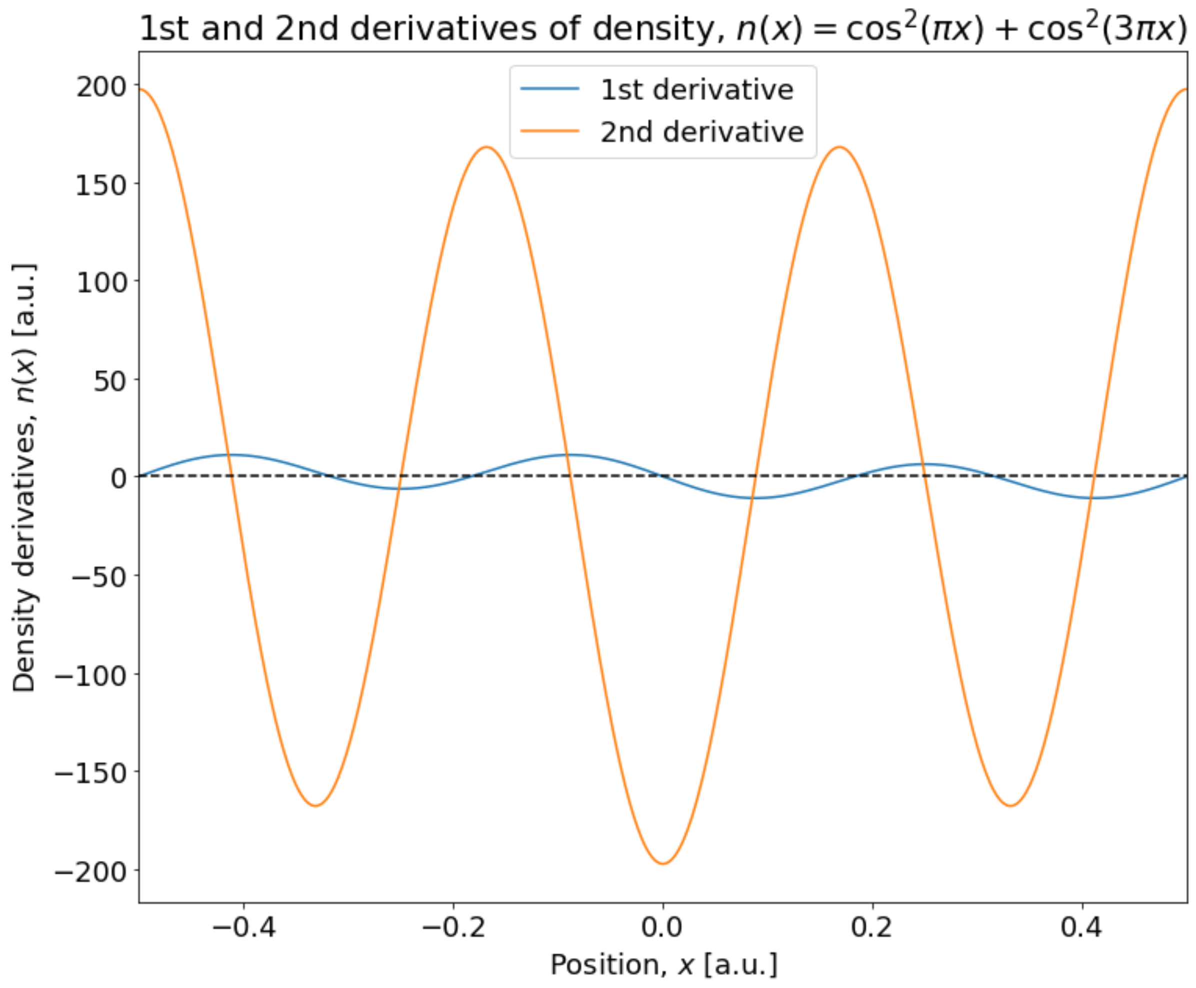
In [145]:
```
def getAnalyticDCarstenDensity( raX, bTestIntegral=False ):
    raDCarstenDensity =  -2.0*np.pi*( np.cos( np.pi*raX )*np.sin( np.pi*
raX ) \
                         +3.0*np.cos( 3.0*np.pi*raX )*np.sin( 3.0*np.pi*
raX ))

    # built-in test checks that the derivative of the density integrates
to zero (by symmetry)
    if( bTestIntegral == True ):
        rDX = raX[1]-raX[0]
        rIntegral = np.sum( raDCarstenDensity ) * rDX
        print('[TEST] Integral of density derivative computed using unif
orm quadrature: ', rIntegral)

    return raDCarstenDensity

def getAnalyticDDCarstenDensity( raX, bTestIntegral=False ):
    raDDCarstenDensity = -2.0*np.pi**2*(np.cos( np.pi*raX )**2-
                                       np.sin( np.pi*raX )**2+
                                       9.0*np.cos( 3.0*np.pi*raX )**2-
                                       9.0*np.sin( 3.0*np.pi*raX )**2)

    # built-in test checks that the second derivative of the density int
egrates to zero,
    # we are integrating a periodic function - without a constant offset
- over a period
    # NOTE: the default of 501 points on the interval won't give a very
 good value for this integral - why not?
    if( bTestIntegral == True ):
        rDX = raX[1]-raX[0]
        rIntegral = np.sum( raDDCarstenDensity ) * rDX
        print('[TEST] Integral of 2nd density derivative computed using
 uniform quadrature: ', rIntegral)

    return raDDCarstenDensity
```

In [146]:
```
raDCarstenDensity = getAnalyticDCarstenDensity( raX, bTestIntegral=True
)
raDDCarstenDensity = getAnalyticDDCarstenDensity( raX, bTestIntegral=Tru
e )
```

```
[TEST] Integral of density derivative computed using uniform quadratur
e:  -2.2737367544323226e-16
[TEST] Integral of 2nd density derivative computed using uniform quadra
ture:  0.39478417604357163
```

In [147]:
```
# plot the density derivatives
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
plot( raX, raDCarstenDensity, label='1st derivative' )
plot( raX, raDDCarstenDensity, label='2nd derivative')

xlabel('Position, $x$ [a.u.]')
ylabel('Density derivatives, $n(x)$ [a.u.]')
title('1st and 2nd derivatives of density, $n(x) = \cos^2(\pi x) + \cos^
2(3\pi x)$')
xlim([rLeft,rRight])
rYLim = 1.1*max(abs(raDDCarstenDensity))
ylim([-rYLim,rYLim])
axhline( 0, color='black', ls='--' )
legend()
```

Out[147]:
```
<matplotlib.legend.Legend at 0x7fee58f67fd0>
```



Zeros of the first and second derivatives

The approximate locations of the zeros should be evident by inspection, from the plot of the density generated in answering Question 1. For rigor's sake, let us write down the analytic forms to gain insight and describe how we might determine these points, numerically.

The expression for the first derivative is,

$$\frac{dn}{dx} = -2\pi\cos(\pi x)\sin(\pi x) - 6\pi\cos(3\pi x)\sin(3\pi x),$$

which is zero when,

$$2\pi\cos(\pi x)\sin(\pi x) = 6\pi\cos(3\pi x)\sin(3\pi x).$$

It is obvious when both sides vanish due to the $\cos(x = \pm 0.5)$ or $\sin(x = 0)$ terms, but it is less obvious when equivalence holds and neither side vanishes. It turns out that there are four more points (two pairs, symmetric about $x = 0$) at which this occurs. These can be verified by the motivated student interested in numerically solving transcendental equations (e.g., using a root finding algorithm).

The expression for the second derivative is,

$$\frac{d^2n}{dx^2} = 2\pi^2\sin^2(\pi x) - 2\pi^2\cos^2(\pi x) + 18\pi^2\sin^2(3\pi x) - 18\pi^2\cos^2(3\pi x).$$

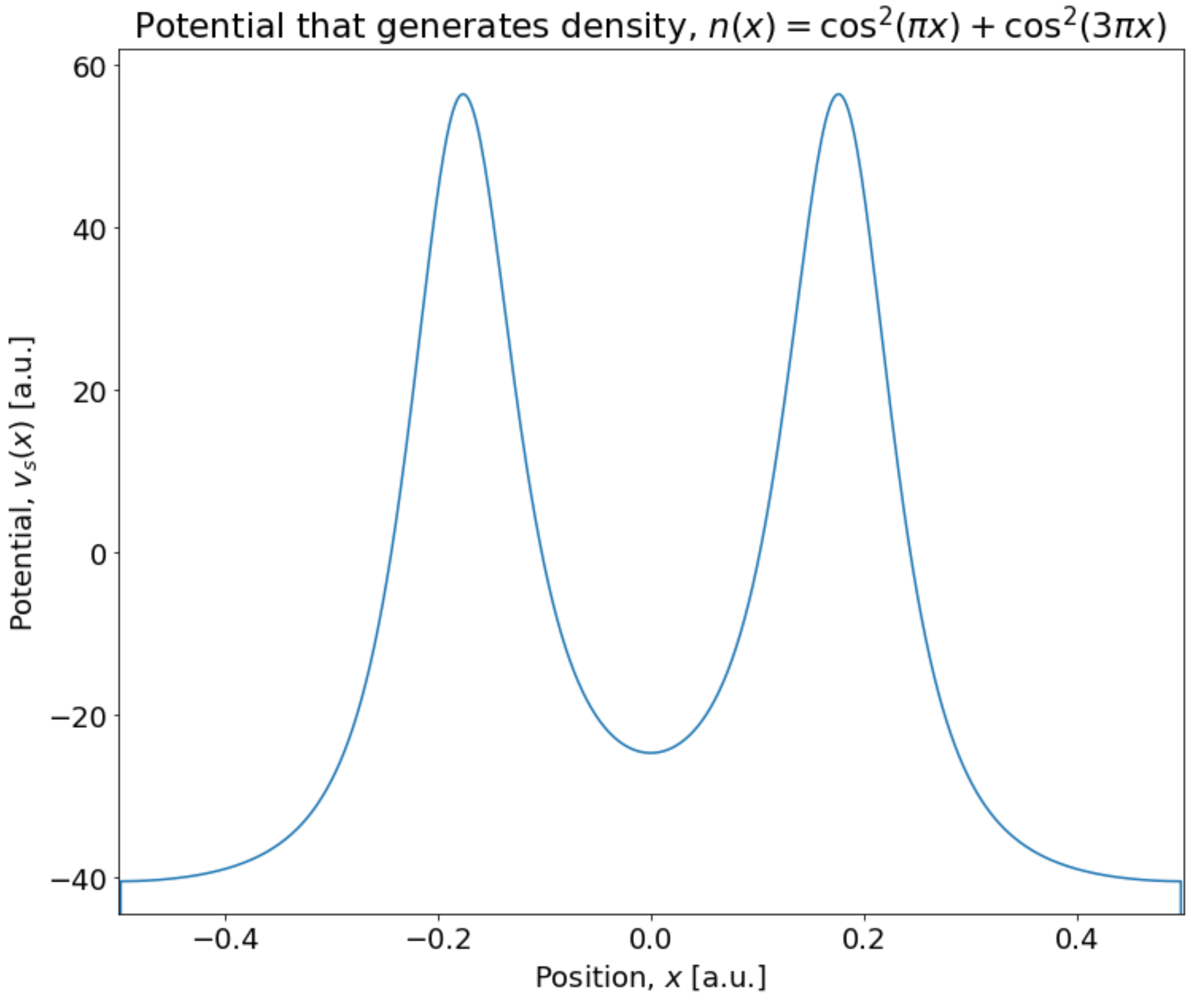
Here, it is even less obvious when this expression vanishes. However, there are two analytically resolvable roots in the interval $[0.5, 0.5]$ at $x = \pm 0.25$. The remaining four zeros can be determined with a root finding algorithm.

In [148]:
```
def getPotential( raN, raDN, raDDN ):
    raPotential=(raDDN/(4.0*raN))-(np.abs(raDN)**2/(8.0*raN**2))
    return raPotential
```

In [149]:
```
# get the density and its derivatives
raN = getCarstenDensity( raX )
raDN = getAnalyticDCarstenDensity( raX )
raDDN = getAnalyticDDCarstenDensity( raX )

# compute the potential
raV = getPotential( raN, raDN, raDDN )
```

In [150]:
```
# plot the potential
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
plot( raX, raV )

xlabel('Position, $x$ [a.u.]')
ylabel('Potential, $v_s(x)$ [a.u.]')
title('Potential that generates density, $n(x) = \cos^2(\pi x) + \cos^2
(3\pi x)$')
xlim([rLeft,rRight])
rYTop = 1.1*max(raV[1:nPoints-1])
rYBot = 1.1*min(raV[1:nPoints-1])
ylim([rYBot,rYTop])
```

Out[150]:
```
(-44.511367199404056, 62.04912064636329)
```



It is evident that there are numerical pathologies in evaluating the potential at the boundaries of the domain. Examining

$$v_s(x) = \frac{\frac{d^2n}{dx^2}}{4n(x)} - \frac{\left(\frac{dn}{dx}\right)^2}{8n(x)^2},$$

we see that this is due to the fact that the density (and likewise, its square) vanishes at $x = \pm 0.5$. The evident divergence arises because we haven't been especially careful about how we evaluate this expression.

We can expand the two terms separately about $x = 0.5$, noting that this is identical by symmetry to $x = -0.5$,

$$\frac{\frac{d^2n}{dx^2}}{4n(x)} \approx \frac{1}{2}(x-0.5)^{-2} - \frac{41\pi^2}{6} + \frac{121\pi^4}{30}(x-0.5)^2 + \mathcal{O}((x-0.5)^4), \text{ and}$$
$$\frac{\left(\frac{dn}{dx}\right)^2}{8n(x)^2} \approx \frac{1}{2}(x-0.5)^{-2} - \frac{41\pi^2}{15} + \frac{413\pi^4}{150}(x-0.5)^2 + \mathcal{O}((x-0.5)^4).$$

We see that the leading order divergent terms cancel each other. The remaining non-constant terms vanish as $x \to 0.5$, leaving only the constants. This gives us $\lim_{x\to\pm 0.5} v_s(x) = -\frac{41\pi^2}{10} \approx -40.465$.

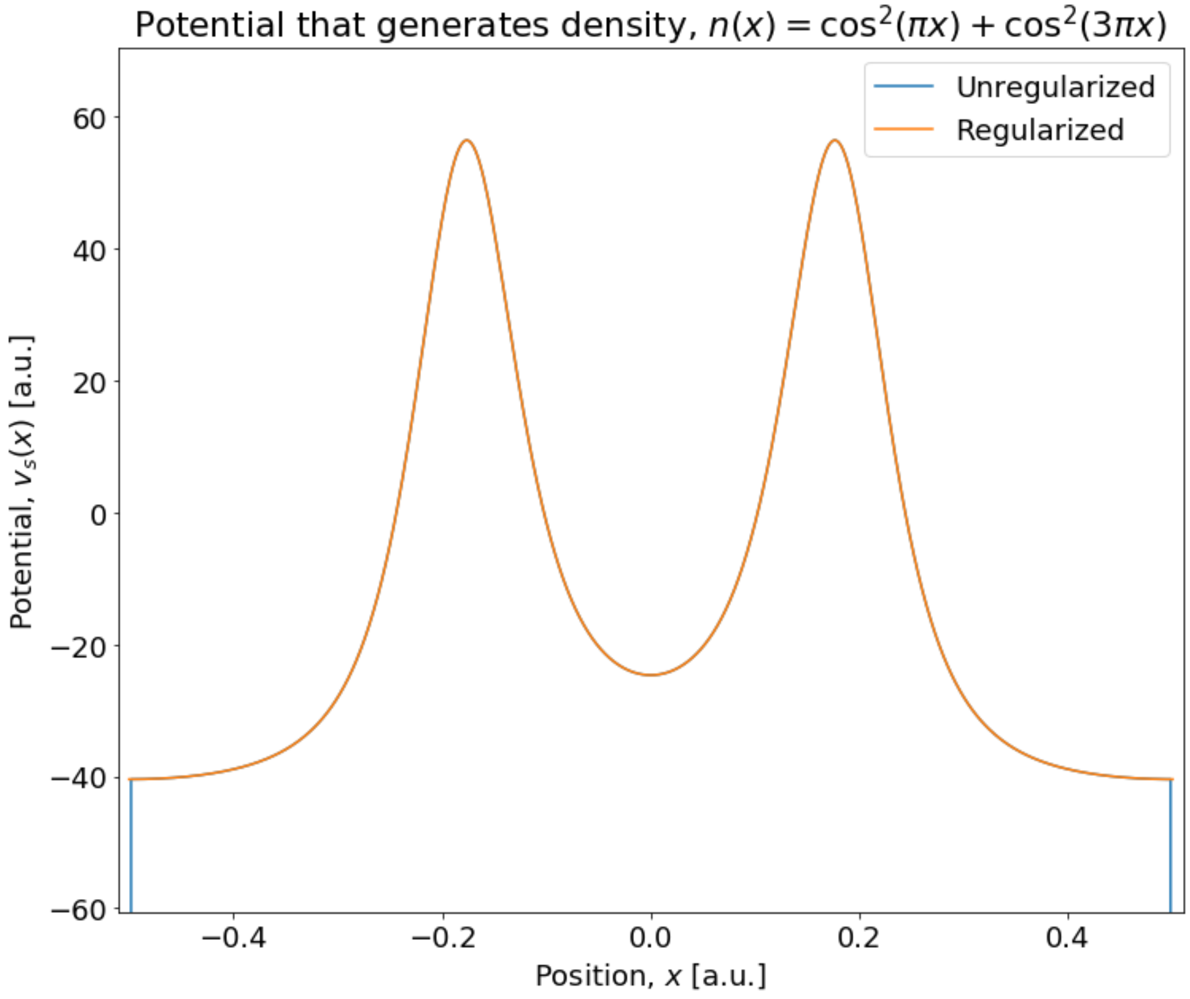
Using this analysis, we can implement a regularized form of the potential that includes a quadratic correction in an epsilon region around the boundary. If you wanted to be **even more** careful, you could interpolate between these two forms.

In [151]:
```
# NOTE: this is one of the least Pythonic things that I've ever construc
ted...but it conveys the idea...
def getPotentialRegularized( raX, rEpsilon=1e-08 ):
    raPotential = np.zeros_like( raX )
    for iIdx, rX in enumerate(raX):
        if( abs(rX)>=0.5-rEpsilon ):
            raPotential[iIdx] = -(41./10.)*np.pi**2 + ((32./25.)*np.pi**
4)*(abs(rX) - 0.5)**2
        else:
            rN = getCarstenDensity( rX )
            rDN = getAnalyticDCarstenDensity( rX )
            rDDN = getAnalyticDDCarstenDensity( rX )
            raPotential[iIdx] = (rDDN/(4.0*rN))-(np.abs(rDN)**2/(8.0*rN*
*2))
    return raPotential
```

In [152]:
```
# compute the regularized potential
raVReg = getPotentialRegularized( raX )
```

In [153]:
```
# plot the potential
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
plot( raX, raV, label='Unregularized')
plot( raX, raVReg, label='Regularized' )

xlabel('Position, $x$ [a.u.]')
ylabel('Potential, $v_s(x)$ [a.u.]')
title('Potential that generates density, $n(x) = \cos^2(\pi x) + \cos^2
(3\pi x)$')
xlim([rLeft,rRight])
rYTop = 1.25*max(raV[1:nPoints-1])
rYBot = 1.5*min(raV[1:nPoints-1])
xlim([-0.51,0.51])
ylim([rYBot,rYTop])
legend()
```

Out[153]:
```
<matplotlib.legend.Legend at 0x7fee58f92b70>
```



Of course, the simplest solution is to just let the value of v_s at the boundary go to its value just interior to the boundary (i.e., within numerical epsilon). However, this more formal solution provides interesting insights into the nature of the numerical pathologies. In this particular case, it turns out that the solution is perfectly well-defined but there are two divergent terms that do not cancel when evaluated separately!

Exercise 4

Numerically verify that the potential you find is indeed the one that solves the associated Schrödinger Equation. Ask yourself,

1. Having the density - and thus, the potential - on a uniform grid, what is the simplest choice of discretization for the Schrödinger equation? (*Note: your idea of "simple" might differ from ours!*)
2. The density goes to zero at the boundary, what should we choose as the boundary condition on $\phi(x)$ at $x = \pm 0.5$?
3. (Bonus) What are other viable choices for the discretization of the Kohn-Sham equation? Are there other possibilities for the boundary condition?

Solution 4

For the purposes of this notebook, we will consider a finite difference discretization of the Kohn-Sham equation to be the "simplest choice". You might disagree, in which case feel free to go with your method of choice!

Recall again that

$$\left[-\frac{1}{2}\frac{d^2}{dx^2} + v_s(x)\right]\phi(x) = \varepsilon\phi(x).$$

A finite difference discretization of this problem will result in a finite-dimensional eigenproblem of the form,

$$\sum_{\beta=1}^{N_g} \mathbf{H}_{\alpha\beta}\phi_n^h(x_\alpha) = \varepsilon_n\phi_n^h(x_\beta) \quad \forall x_\alpha \in \{x_0, \dots, x_{N_g+1} | x_\alpha = -0.5 + \frac{\alpha}{N_g}\},$$

where we have chosen the indexing for our grid points to be conscientious of the fact that we will remove the two boundary nodes at $x = \pm 0.5$ from the problem. We have used the notation $\phi_n^h(x)$ to indicate that $\phi_n^h(x) \approx \phi_n(x) + \mathcal{O}(h^2)$, where $h = N_g^{-1}$ is the grid spacing.

The only non-trivial part of constructing $\mathbf{H}_{\alpha\beta}$ is approximating the second derivative operator with a finite difference stencil. That is,

$$-\frac{1}{2}\frac{d^2}{dx^2}\phi_n(x_\alpha) \approx -\frac{1}{2}\frac{\phi_n^h(x_{\alpha-1}) - 2\phi_n^h(x_\alpha) + \phi_n^h(x_{\alpha+1})}{h^2}.$$

Because the density goes to zero at $x = \pm 0.5$, we can enforce that $\phi_n^h(x_0) = \phi_n^h(x_{N_g+1}) = 0$ (i.e., a *homogeneous Dirichlet boundary condition*). We can then eliminate the rows and columns of $\mathbf{H}_{\alpha\beta}$ that correspond to the orbital's values at these points (i.e., they are known). Thus, we have chosen the indexing scheme to go from 0 to $N_g + 1$, because there are still only N_g equations in N_g unknowns. The remaining trivial portion of $\mathbf{H}_{\alpha\beta}$ is simply a diagonal contribution corresponding to v_s such that,

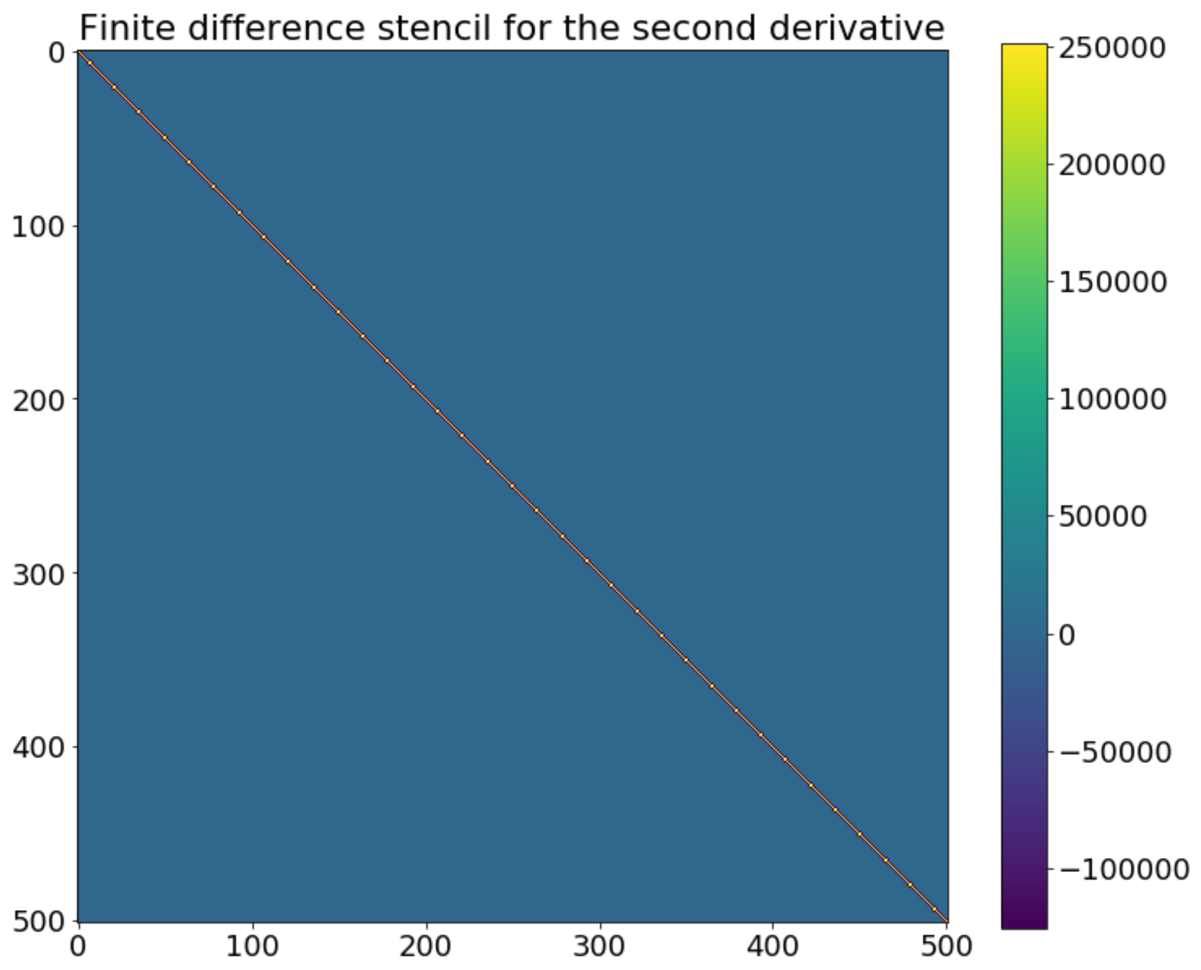
$$\mathbf{H}_{\alpha\beta} = -\frac{\nabla_{\alpha\beta}}{2} + v_s(x_\alpha)\delta_{\alpha\beta},$$

where $\nabla_{\alpha\beta}$ encodes the finite difference stencil defined above and $\delta_{\alpha\beta}$ is the Kronecker delta.

Note that the matrix, $\mathbf{H}_{\alpha\beta}$, is especially sparse. It is what is known as a tridiagonal matrix. This type of matrix holds a special place in numerical linear algebra - the first step in many eigensolvers for Hermitian matrices is the reduction to a tridiagonal form. Here, we are cutting out that step whether we care to appreciate it or not. The example code below makes use of sparse data types from scipy.sparse.

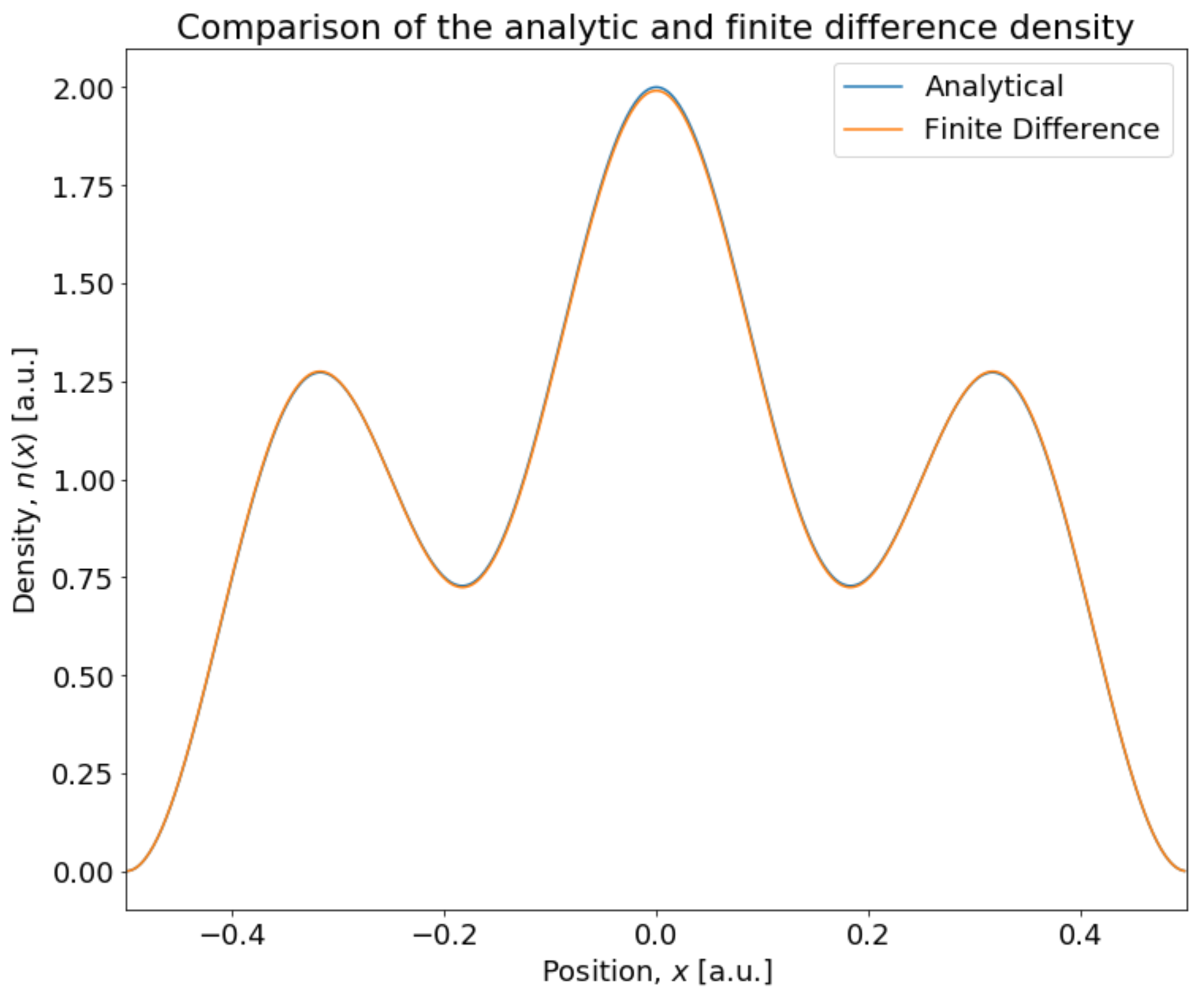
In [154]:
```
# re-define the grid to account for the "new" indexing scheme
nNg = 501
raX_fd = np.linspace(-0.5, 0.5, nNg+2)

# define the second derivative term (complete with the factor of -1/2)
sparsematD2 = sps.diags( [-nNg**2/2., nNg**2, -nNg**2/2.], [-1,0,1], sha
pe=(nNg,nNg) )
figure(figsize=[12,10])
mpl.rcParams['font.size'] = 18
imshow(sparsematD2.todense())
title('Finite difference stencil for the second derivative')
colorbar()
```

Out[154]:
```
<matplotlib.colorbar.Colorbar at 0x7fee0b451a90>
```



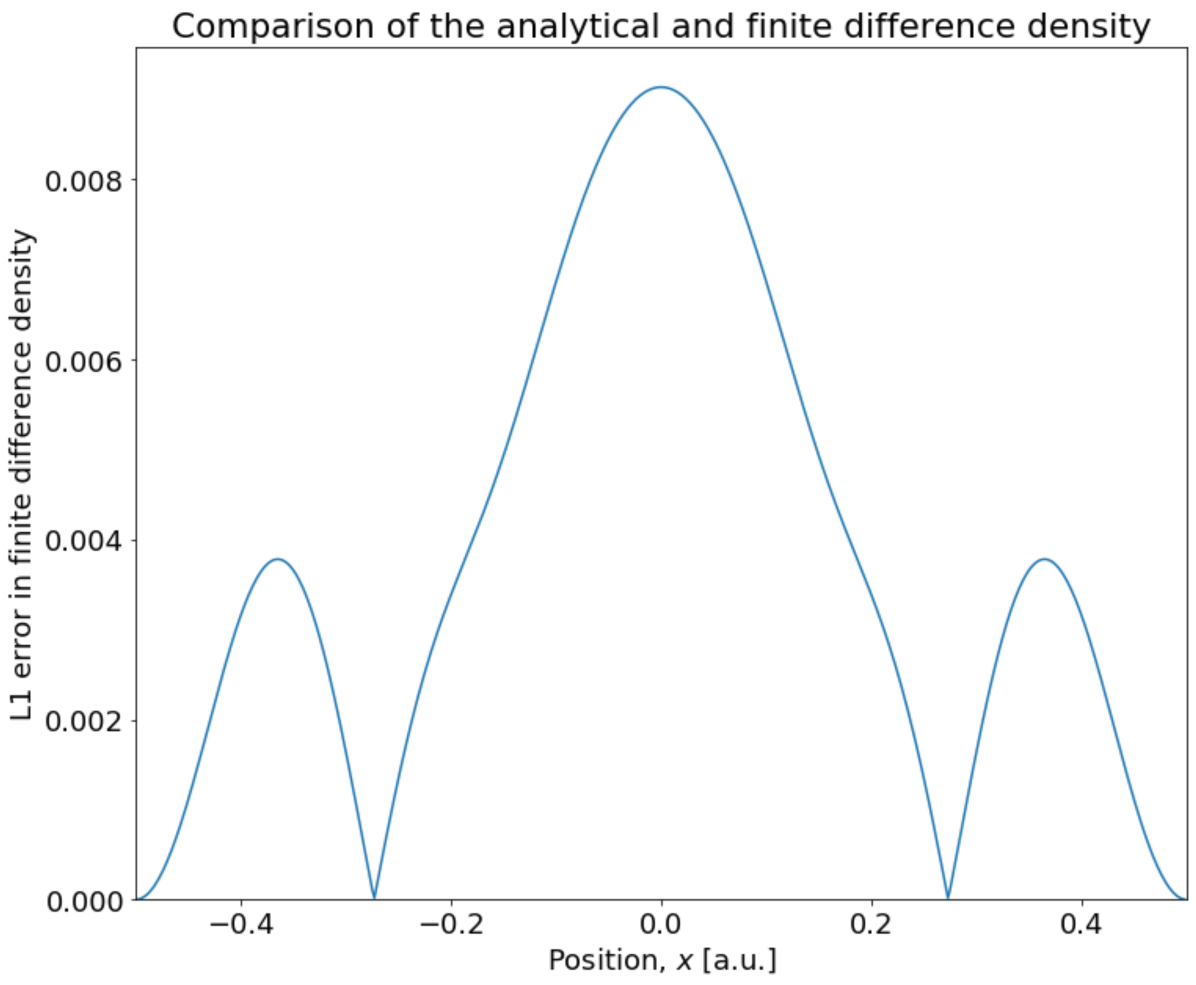
In [155]:
```
# next, fill out a diagonal matrix with the potential term
sparsematVs = sps.diags( [getPotentialRegularized( raX_fd[1:nNg+1] )], [
0], shape=(nNg,nNg) )

# add the two terms together
sparsematH = sparsematD2+sparsematVs

# set the number of states to resolve in the eigensolve
nOrbitals = 5

# apply a sparse eigensolver
raEnergies, matSolutions = spsl.eigsh( sparsematH, k=nOrbitals, which='S
A' )

# the eigenvectors come out with unit norm, i.e., sum(solution**2)=1
# we want sum(orbital**2)*h=1 to get the right units, though
# so, orbital = solution/sqrt(h)

# compute the normalized orbitals
matOrbitals = np.zeros([nNg,nOrbitals])
for iIdx in np.arange(nOrbitals):
    matOrbitals[:,iIdx] = matSolutions[:,iIdx]*np.sqrt(nNg)
```

```
In [156]: # plot the ground state density computed from the normalized orbitals ob
          tain in the finite difference calculation
          figure( figsize=[12,10] )
          mpl.rcParams['font.size'] = 18
          plot( raX_fd[1:nNg+1], getCarstenDensity( raX_fd[1:nNg+1]), label='Analy
          tical')
          plot( raX_fd[1:nNg+1], matOrbitals[:,0]**2, label='Finite Difference' )

          xlabel('Position, $x$ [a.u.]')
          ylabel('Density, $n(x)$ [a.u.]')
          title('Comparison of the analytic and finite difference density')
          xlim([-0.5,0.5])
          legend()

Out[156]: <matplotlib.legend.Legend at 0x7fee28dd6668>
```



In [157]:
```
# plot the ground state density computed from the normalized orbitals ob
tain in the finite difference calculation
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
plot( raX_fd[1:nNg+1], abs(getCarstenDensity( raX_fd[1:nNg+1])-matOrbita
ls[:,0]**2) )

xlabel('Position, $x$ [a.u.]')
ylabel('L1 error in finite difference density')
title('Comparison of the analytical and finite difference density')
xlim([-0.5,0.5])
ylim(bottom=0)
```

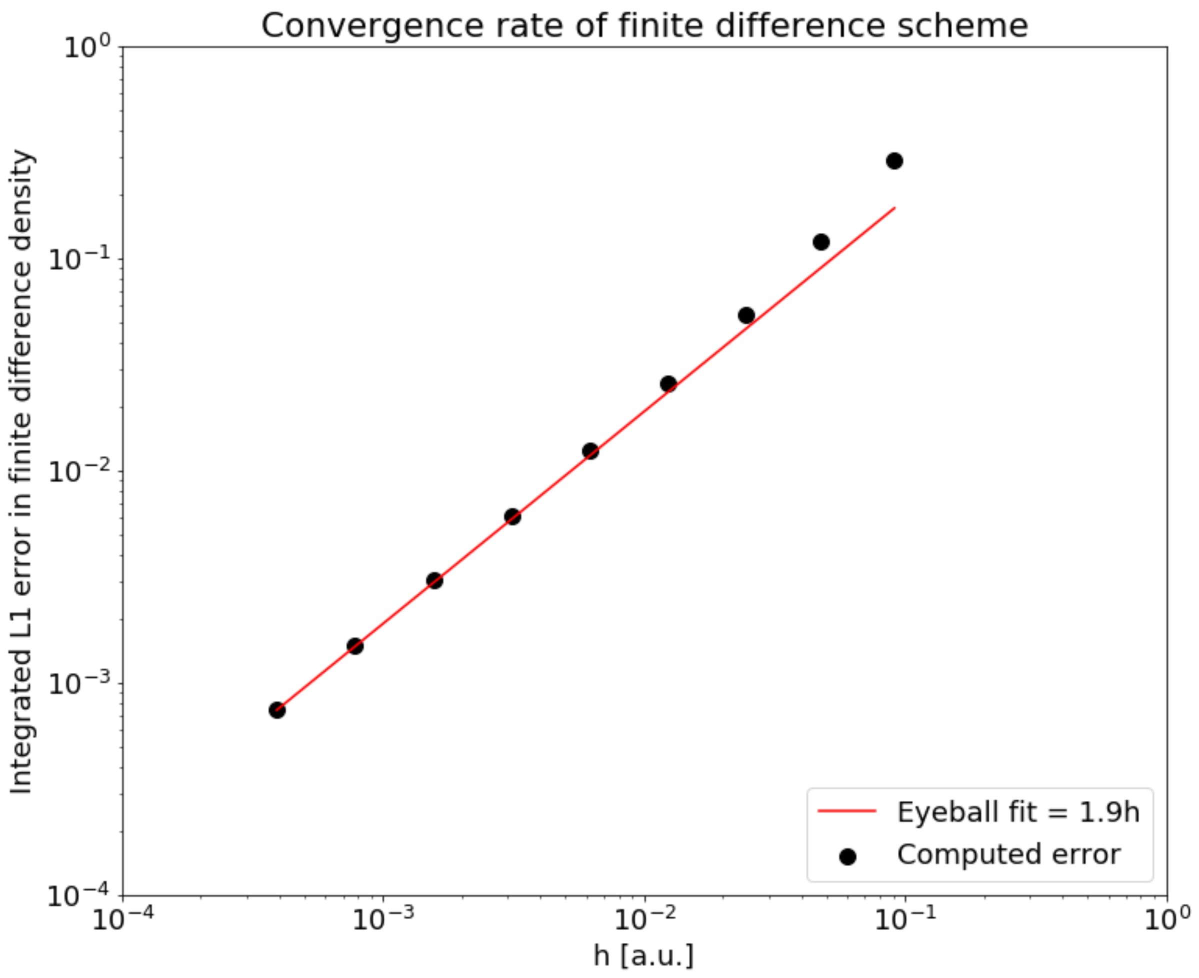
Out[157]:
```
(0, 0.009474408562763917)
```



In [158]:
```
# test the convergence rate of the finite difference calculation
naNg = np.array([11,21,41,81,161,321,641,1281,2561])
raError = np.zeros(len(naNg))

# loop over different values of h and compute an integrated error
for iIdx, nNg in enumerate(naNg):
    raX_fd = np.linspace(-0.5, 0.5, nNg+2)
    sparsematD2 = sps.diags( [-nNg**2/2., nNg**2, -nNg**2/2.], [-1,0,1],
shape=(nNg,nNg) )
    sparsematVs = sps.diags( [getPotentialRegularized( raX_fd[1:nNg+1]
)], [0], shape=(nNg,nNg) )
    sparsematH = sparsematD2+sparsematVs
    nOrbitals = 10
    raEnergies, matSolutions = spsl.eigsh( sparsematH, k=nOrbitals, whic
h='SA' )
    matOrbitals = np.zeros([nNg,nOrbitals])
    for iJdx in np.arange(nOrbitals):
        matOrbitals[:,iJdx] = matSolutions[:,iJdx]*np.sqrt(nNg)
    raError[iIdx] = np.sum(abs(getCarstenDensity( raX_fd[1:nNg+1])-matOr
bitals[:,0]**2))/nNg
```

In [159]:
```
# plot the "error" as a function of h
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
scatter( 1./naNg, raError, s=100, color='black', label='Computed error'
)
plot( 1./naNg, 1.9/naNg, color='red', label='Eyeball fit = 1.9h' )

xlabel('h [a.u.]')
ylabel('Integrated L1 error in finite difference density')
title('Convergence rate of finite difference scheme')
xlim([0.0001,1.0])
ylim([0.0001,1.0])
xscale('log')
yscale('log')
legend(loc=4)
```

Out[159]:
```
<matplotlib.legend.Legend at 0x7fee19294198>
```



The convergence rate in our contrived integrated error metric is linear in the grid spacing.

Question 5

For one-dimensional problems, it is convenient to consider either contact ($v_{int}(x,x') = \delta(x-x')$) or regularized Coulomb interactions ($v_{int}(x,x') = ((x-x')^2 + c^2)^{-1/2}$). Choose a "flavor" of interaction and implement subroutines that compute:

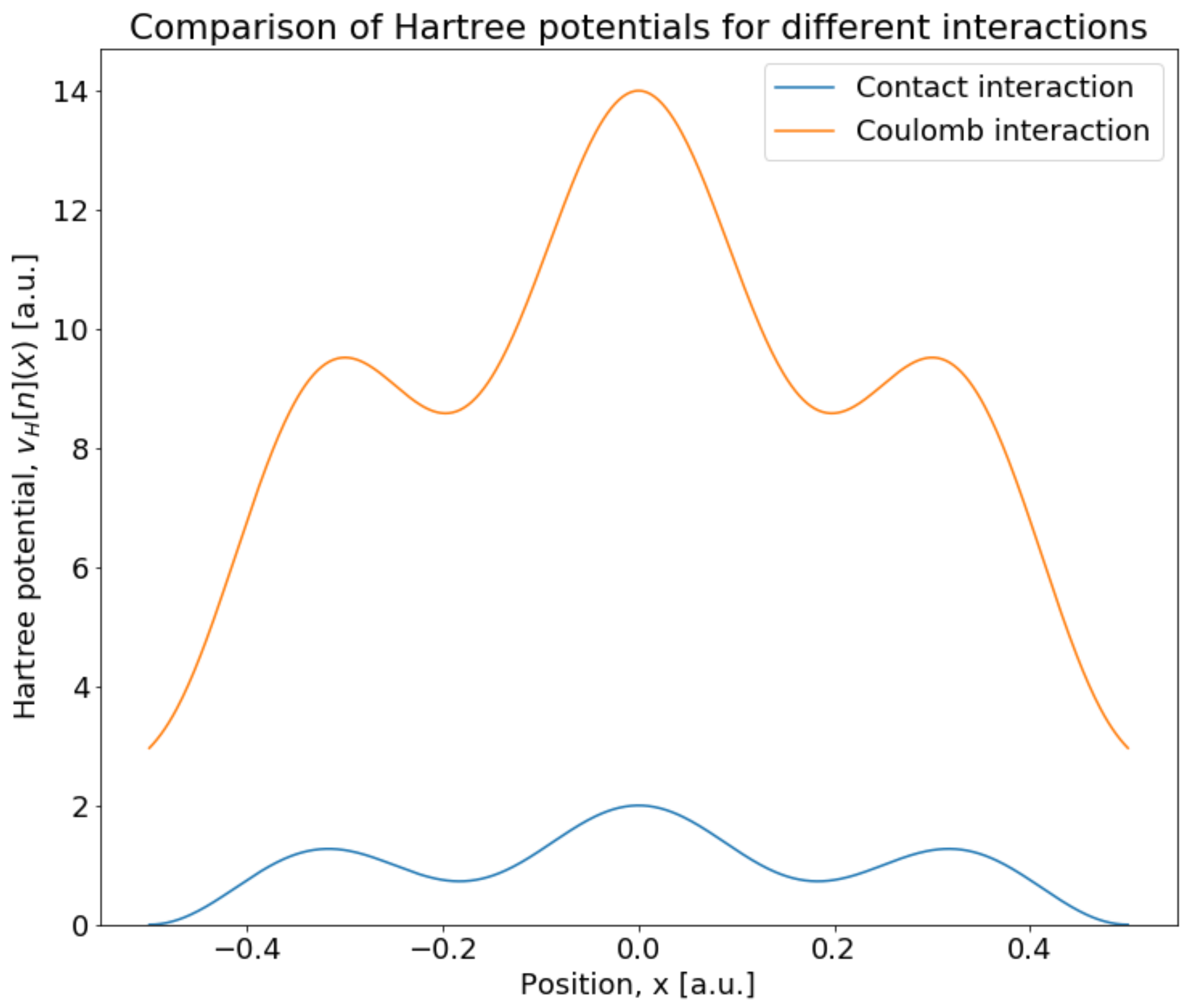
1. The Hartree potential given a density,
$$v_H[n](x) = \int dx' \, v_{int}(x,x')n(x').$$
2. The total Hartree energy given a Hartree potential,
$$E_H[n] = \int dx v_H[n](x)n(x).$$
3. (Bonus) The elements of the four-index tensor given four orbitals,
$$V_{ijkl} = \int dx \int dx' \, v_{int}(x,x')\phi_i(x)\phi_j(x)\phi_k(x')\phi_l(x')$$

Solution 5

In [160]:
```
def getVHartreeDelta( raX, raN ):
    raVHartree = np.zeros_like( raN )
    raVHartree = raN
    return raVHartree

def getVHartreeCoulomb( raX, raN, rC=0.01 ):
    raVHartree = np.zeros_like( raN )
    rH = raX[1]-raX[0]
    for iIdx, rX in enumerate(raX):
        raVHartree[iIdx] = np.sum( raN[:]/sqrt((raX[:]-rX)**2+rC**2) )*r
H
    return raVHartree
```

In [161]:
```
# plot the Hartree potentials from Carsten's density
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18
plot( raX, getVHartreeDelta(raX, raN), label='Contact interaction')
plot( raX, getVHartreeCoulomb(raX, raN), label='Coulomb interaction')

xlabel('Position, x [a.u.]')
ylabel('Hartree potential, $v_H[n](x)$ [a.u.]')
title('Comparison of Hartree potentials for different interactions')
legend()
ylim(bottom=0)
```

Out[161]:
```
(0, 14.690829016122692)
```



TBD: adding the four-index matrix elements

I anticipate that this will be useful for motivated students aiming to develop a CI calculation from which they could do, e.g., an inversion!

Question 6

Set up a simple time-dependent KS-DFT propagator using your method of choice. This could include explicit time stepping schemes (e.g., Runge-Kutta or a Taylor expansion) or implicit time stepping schemes (e.g., Crank-Nicolson). Ask yourself,

1. What is the most elementary subroutine that I need to do *any* time propagation scheme? *Hint: it involves applying an operator to each orbital*
2. What numerical linear algebra routines do you need for your approach?
3. What advantages and disadvantages does your scheme have relative to the others? Is it unitary? What is the order of convergence?

Solution 6

First, we implement a low-storage version of 4th order Runge-Kutta from Carpenter and Kennedy, https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19940028444.pdf (https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19940028444.pdf).

Because it is a low-storage scheme, we only need to store $2N_gN_{orb}$ coefficients at once. It is not a unitary propagator and it does not conserve the norm of the orbitals, so we slowly "leak" charge.

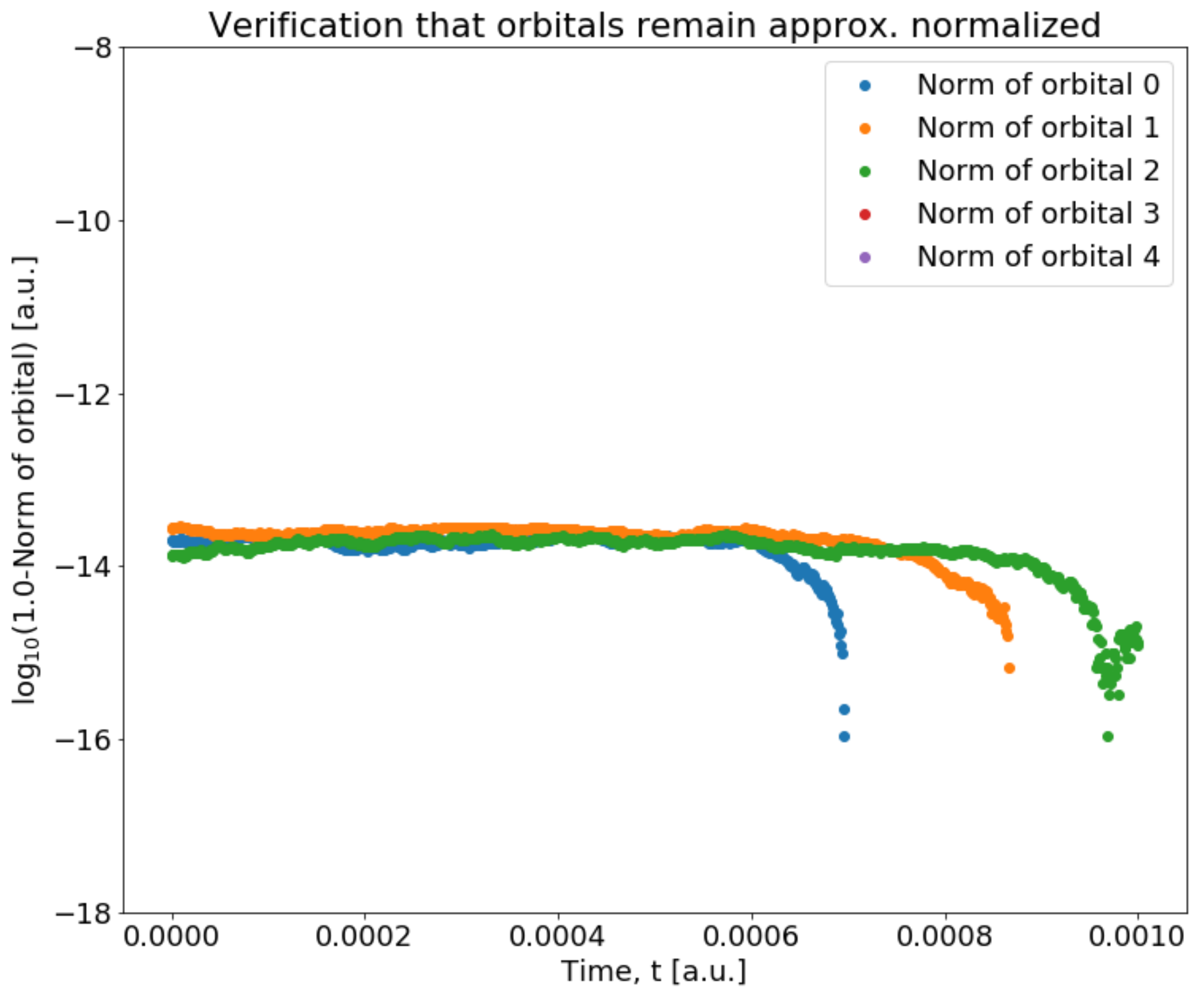
In [211]:
```
# set the A, B, and C coefficients for each stage
raRKA = np.array([ 0.0, \
                  -567301805773.0/1357537059087.0, \
                  -2404267990393.0/2016746695238.0, \
                  -3550918686646.0/2091501179385.0, \
                  -1275806237668.0/842570457699.0], dtype=np.float64)
raRKB = np.array([ 1432997174477.0/9575080441755.0, \
                   5161836677717.0/13612068292357.0, \
                   1720146321549.0/2090206949498.0, \
                   3134564353537.0/4481467310338.0, \
                   2277821191437.0/14882151754819.0], dtype=np.float64)
raRKC = np.array([ 0.0, \
                   1432997174477.0/9575080441755.0, \
                   2526269341429.0/6820363962896.0, \
                   2006345519317.0/3224310063776.0, \
                   2802321613138.0/2924317926251.0], dtype=np.float64)

# write a subroutine that advances the orbitals in zaOrbitals by rDt
def stepLSRK4( zaOrbitals, rTime, rDt, sparsematH, getRHS, **kwargs ):

    zaDOrbitals = np.zeros_like( zaOrbitals )

    for iStage in np.arange(5):

        rA = raRKA[iStage]
        rB = raRKB[iStage]
        rC = raRKC[iStage]

        rIntrastepTime = rTime + rDt*rC

        zaRHS = getRHS( zaOrbitals, rIntrastepTime, sparsematH, **kwargs
)
        zaDOrbitals = rA*zaDOrbitals + rDt*zaRHS
        zaOrbitals = zaOrbitals + rB*zaDOrbitals

    return zaOrbitals

def getRHS_constantHamiltonian( zaOrbitals, rIntrastepTime, sparsematH
):

    zaRHS = np.zeros_like( zaOrbitals )
    for iOrbital, zaOrbital in enumerate(zaOrbitals.T):
        zaRHS[:,iOrbital] = -1j*sparsematH.dot(zaOrbital)
    return zaRHS
```

In [245]:
```
# create a Hamiltonian and diagonalize it to generate orbitals for an in
itial condition
nNg = 501
raX_fd = np.linspace(-0.5, 0.5, nNg+2)
sparsematD2 = sps.diags( [-nNg**2/2., nNg**2, -nNg**2/2.], [-1,0,1], sha
pe=(nNg,nNg) )
sparsematVs = sps.diags( [getPotentialRegularized( raX_fd[1:nNg+1] )], [
0], shape=(nNg,nNg) )
sparsematH = sparsematD2+sparsematVs
nOrbitals = 5
raEnergies, matSolutions = spsl.eigsh( sparsematH, k=nOrbitals, which='S
A' )
matOrbitals = np.zeros([nNg,nOrbitals])
for iJdx in np.arange(nOrbitals):
    matOrbitals[:,iJdx] = matSolutions[:,iJdx]*np.sqrt(nNg)

# load the first 5 orbitals as an initial condition
zaOrbitals = matOrbitals.astype(complex)

# set up the number of time steps and the associated temporal grid
nT = 1000
raTime = np.linspace( 0.0, 0.001, nT)
rDt = raTime[1]-raTime[0]

# record the norm of the orbitals at each time step
raNorm = np.zeros([nT,nOrbitals], dtype=np.float64)
for iOrbital, zaOrbital in enumerate(zaOrbitals.T):
    raNorm[0,iOrbital] = np.real((np.dot(zaOrbital,zaOrbital)/nNg))

# time propagate!
for iTime, rTime in enumerate(raTime[:-1]):

    zaOrbitals = stepLSRK4( zaOrbitals, rTime, rDt, sparsematH, getRHS_c
onstantHamiltonian )
    for iOrbital, zaOrbital in enumerate(zaOrbitals.T):
        raNorm[iTime+1,iOrbital] = np.real((np.vdot(zaOrbital,zaOrbital)
/nNg))
```

In [272]:
```
# plot the norm of iOrbital
iOrbital = 0
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18

for iOrbital in np.arange(nOrbitals):
    plot( raTime, log10(1.0-raNorm[:,iOrbital]), 'o', label='Norm of orb
ital '+str(iOrbital))

xlabel('Time, t [a.u.]')
ylabel('$\log_{10}$(1.0-Norm of orbital) [a.u.]')
title('Verification that orbitals remain approx. normalized')
ylim([-18,-8])
legend()
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:7: Runtime
Warning: invalid value encountered in log10
  import sys
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:7: Runtime
Warning: divide by zero encountered in log10
  import sys
```

Out[272]:
```
<matplotlib.legend.Legend at 0x7fee3b832e80>
```



Next, we implement a Crank-Nicolson time stepping scheme.

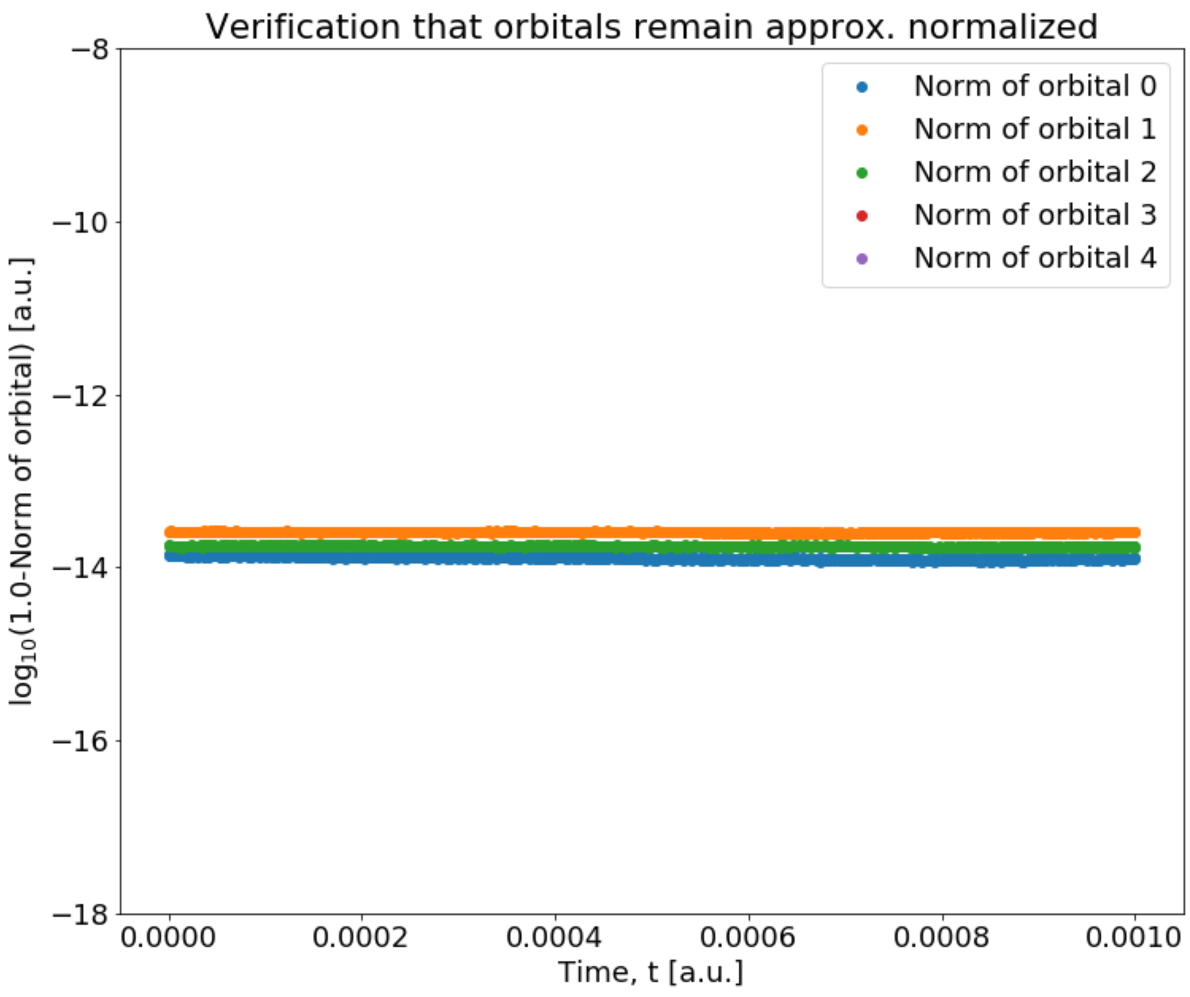
In [273]:
```
# write a subroutine that advances the orbitals in zaOrbitals by rDt
def stepCN( zaOrbitals, rTime, rDt, sparsematH, getRHS, rTol=1e-8, **kwa
rgs ):

    sparsematCN = sps.identity( zaOrbitals.shape[0] ) + 0.5*1j*rDt*spars
ematH
    for iOrbital, zaOrbital in enumerate(zaOrbitals.T):
        zaRHS = (sparsematCN.conjugate()).dot(zaOrbital)
        zaOrbitals[:,iOrbital], iInfo = spsl.gmres( sparsematCN, zaRHS,
tol=rTol )
        if( iInfo!=0 ):
            print('Problem in GMRES!')

    return zaOrbitals
```

In [276]:
```
# create a Hamiltonian and diagonalize it to generate orbitals for an in
itial condition
nNg = 501
raX_fd = np.linspace(-0.5, 0.5, nNg+2)
sparsematD2 = sps.diags( [-nNg**2/2., nNg**2, -nNg**2/2.], [-1,0,1], sha
pe=(nNg,nNg) )
sparsematVs = sps.diags( [getPotentialRegularized( raX_fd[1:nNg+1] )], [
0], shape=(nNg,nNg) )
sparsematH = sparsematD2+sparsematVs
nOrbitals = 5
raEnergies, matSolutions = spsl.eigsh( sparsematH, k=nOrbitals, which='S
A' )
matOrbitals = np.zeros([nNg,nOrbitals])
for iJdx in np.arange(nOrbitals):
    matOrbitals[:,iJdx] = matSolutions[:,iJdx]*np.sqrt(nNg)

# load the first 5 orbitals as an initial condition
zaOrbitals = matOrbitals.astype(complex)

# set up the number of time steps and the associated temporal grid
nT = 1000
raTime = np.linspace( 0.0, 0.001, nT)
rDt = raTime[1]-raTime[0]

# record the norm of the orbitals at each time step
raNorm = np.zeros([nT,nOrbitals], dtype=np.float64)
for iOrbital, zaOrbital in enumerate(zaOrbitals.T):
    raNorm[0,iOrbital] = np.real((np.dot(zaOrbital,zaOrbital)/nNg))

# time propagate!
for iTime, rTime in enumerate(raTime[:-1]):

    zaOrbitals = stepCN( zaOrbitals, rTime, rDt, sparsematH, getRHS_cons
tantHamiltonian, rTol=1e-16 )
    for iOrbital, zaOrbital in enumerate(zaOrbitals.T):
        raNorm[iTime+1,iOrbital] = np.real((np.vdot(zaOrbital,zaOrbital)
/nNg))
```

In [277]:
```
# plot the norm of iOrbital
iOrbital = 0
figure( figsize=[12,10] )
mpl.rcParams['font.size'] = 18

for iOrbital in np.arange(nOrbitals):
    plot( raTime, log10(1.0-raNorm[:,iOrbital]), 'o', label='Norm of orb
ital '+str(iOrbital))

xlabel('Time, t [a.u.]')
ylabel('$\log_{10}$(1.0-Norm of orbital) [a.u.]')
title('Verification that orbitals remain approx. normalized')
ylim([-18,-8])
legend(loc=1)
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:7: Runtime
Warning: invalid value encountered in log10
  import sys
```

Out[277]:
```
<matplotlib.legend.Legend at 0x7fee2a5ebcf8>
```



In [ ]:

In [ ]: