MADNESS

Multiresolution
Adaptive
Numerical
Scientific
Simulation

- *Oak Ridge National Laboratory, and the University of Tennessee, Knoxville*

  George Fann, Robert J. Harrison,

  Zhengting Gan, Takeshi Yanai

- *Colorado University, Boulder*

  Gregory Beylkin, Fernando Perez,

  Lucas Monzon

- *Ohio University, Athens*

  Martin Mohlenkamp

An Introduction to MADNESS

- What is it?
- Simple example of use
- The Function class
- Another example and extending with autoswig
- Multiresolution representation of functions
- How does the S/W work?
- More detailed examples
- Separated representation of operators
- Examples using operators

What is MADNESS?

- Multiresolution Adaptive Numerical Scientific Simulation
  - It is an environment to prototype and develop scientific applications using multiresolution methods in multiwavelet basis
  - The important key words/phrases are
    - Multiresolution, multiwavelet, low-separation rank
  - It presently contains just a Density Functional Theory (DFT) code, but others are working on fluid dynamics, climate modeling, …

Objectives

- Fast algorithms with guaranteed precision
  - Accomplished via a sound mathematical foundation
- High-level composition of numerical codes
  - Work with functions and operators rather than with sparse arrays of matrix elements
  - Matlab-like environment
- Eliminate basis set error
  - AO basis far from complete for most operators

How to use it?

- E.g., Generate a numerical representation of a simple function and print its value out at a point

```
from mra import *

def testf(x,y,z):
    x, y, z = x-0.5, y-0.5, z-0.5
    return exp(-50.0*(x*x+y*y+z*z))

f = Function(function=testf,thresh=1e-3,k=5)
print “Numerical value: %.4f" % f(0.33,0.44,0.55)
print " Analytic value: %.4f" % testf(0.33,0.44,0.55)
```

Line-by-line analysis

```
from mra import *
```

- “import” is Python’s term for including functionality and defintions from other modules.
- The most important thing in the “mra” module is the Function class which supports 3D numerical functions. There are also 1D, 2D, 6D Function classes and also operators to act on these functions.

Line-by-line example I - cont'd

```
def testf(x,y,z):
    x, y, z = x-0.5, y-0.5, z-0.5
    return exp(-50.0*(x*x+y*y+z*z))
```

- This defines a Python procedure for $e^{-50r**2}$.
- The numerical functions are defined on [0,1]
  - the "simulation" cell or frame
- The user cell is most usually [-∞, ∞] so you have to truncate to a finite cell [-L/2,L/2] and then shift and scale coordinates.

Line-by-line example I - cont'd

```
f = Function(function=testf,
             thresh=1e-3,k=5)
```

- Forms a numerical representation of the Python function "testf" with a 5'th order wavelet and adaptively refining to provide a norm-wise precision of 1e-3.

Line-by-line example I - cont'd

```
print "Numerical value: %.4f" % \
  f(0.33,0.44,0.55)
print " Analytic value: %.4f" % \
  testf(0.33,0.44,0.55)
```

- The numerical functions behave much like algebraic functions, but with only finite precision.
- They can be evaluated by calling them with the desired arguments.

The Function Class

```
Function(function=None, cfunction=None, thresh=1e-7,
         k=9, initial_level=2, refine=1, compress=1):
  calling as a Function to evaluate
  +, -, * with scalars and numerical or python functions
  squaring with **2 notation
  reconstruct(nonstandard=0):
  compress(truncate=0):
  truncate(thresh=None):
  project(newk):
  norm2():
  inner(other):
  trace():
  diff(axis,transpose=0):  (note Dx, Dy, Dz)
  diff2(axis):
  laplacian(): (note delsq)
  local_function(function=None,cfunction=None):
  mul_func(function=None,cfunction=None):
  scale(alpha):
  gaxpy(alpha, other, beta):
  griddx(filename='/tmp/grid.dx'):
  plotdx(filename='/tmp/plot.dx', npt=(20,20,20),
         box=(0,1,0,1,0,1)):
```

```
Function(function=None, cfunction=None,
     thresh=1e-7, k=9, initial_level=2,
     refine=1, compress=1)
```

- function specifies a Python callable function
- cfunction specifies a pointer to a C callable function (can specify one or both)
- thresh is the truncation threshold
- k is the order of the wavelet. For efficiency of integral operators, usually choose k to satisfy thresh=10**(2-k).
- initial_level=2 is the level at which the function is first sample for adaptive refinement
- refine=true/false enables adaptive refinement
- compress=true/false compresses the function
- can override defaults by changing the class

- Calling as a Function to evaluate
- Seen this already
  - E.g.,

```
f = Function(function=testf)
print f(0.4,0.3,0.2)
```

  - Note that the function is defined on the unit cube [0,1] … so you must transform from your coordinates (e.g., the molecular coordinates) to the simulation frame.
  - If you are going to evaluate at many points, first reconstruct the function (see below)
  - If you need even greater speed, we can implement a much faster operation to evaluate at many points at the same time.

- Basic arithmetic operations (+, -, *) are supported for combination with real scalars, other MRA functions, and Python functions
- E.g.,

```
f = 2.0*f
f = 2.0*g + 3.0*f
f = h-pi*g
f = g*h + f
```

where f, g, and h are functions

(if a Python function is given it is quietly and expensively first turned into a numerical form)

- Addition and subtraction of functions happen rapidly and accurately.
- The multiplication operation deserves special mention
  - Both functions are first reconstructed (below)
  - Then the sum coefficients of each function are transformed down to the locally lowest level
  - If autorefine is enabled, they are refined one more level down
  - Then, the value of the functions are evaluated on the Legendre quadrature grid, the multiplication performed, and the result transformed
  - Thus, a loss of precision is inherent in the process.
  - Enabling autorefine provides more (almost always adequate) precision but is 8x more expensive.
  - Without autorefine is often adequate (and is used for the DFT, though some care is needed with GGA)

- Squaring is a special case of multiplication and writing `f**2` will be faster than `f*f`

`reconstruct(nonstandard=0)`: `compress(truncate=0)`:

- Move between the representation in the scaling function basis (reconstructed) and in the wavelet basis (compressed)
- In general the reconstructed form is dense, whereas the compressed form is sparse.
- The non-standard form is used to prepare for application of an integral operator.
- Operate in place.

`truncate(thresh=None):`

- Truncates small difference (wavelet) coefficients
- If called, and the wavefunction is not already compressed, compress() is first called.
- Operate in place.

`project(newk):`

- Returns a new function that is formed by crudely projecting the function into the basis of order newk.
- The function is reconstructed, then
  - if newk>=oldk, sum coefficients are copied
    --- no precision is lost
  - if newk<oldk, sum coefficients for polynomials order <= newk are copied and others are discarded
    --- precision is lost

`norm2()`:

- `f.norm2()` returns $\|f\|_2$

`inner(other)`:

- `f.inner(g)` returns

$$(f.g) = \int f(x,y,z)g(x,y,z)dxdydz$$

`trace()`:

- `f.trace()` returns $\int f(x,y,z)dxdydz$

`diff(axis,transpose=0):`

- f.diff(axis) returns new function containing the derivative. axis=0,1,2. Note can also use `Dx*f` or `Dy*f` or `Dz*f` or `DxT*f` etc.

`diff2(axis):`

- Returns a new function with the second derivative. Can use D2x*f, etc. Same as (Dx*DxT+DxT*Dx)/2

`laplacian():`

- Returns a new function as from Delsq*f

`del():`

- Returns the tuple (Dx*f,Dy*f,Dz*f)

- Note that some care is needed to form a satisfactory matrix representation of the Laplacian
- Our basis set is discontinuous, so derivatives are only defined in a weak sense (i.e., as a projection back into the basis) and higher derivatives have more numerical noise.
- Integration by parts shows that

$$\int_0^1 f(x)\frac{d^2 g(x)}{dx^2}dx = \left[\frac{df}{dx}\frac{dg}{dx}\right]_0^1 - \int_0^1 \frac{df(x)}{dx}\frac{dg(x)}{dx}dx$$

- If (as is the case for DFT) our functions and their derivatives vanish at the end points, only the second term survives
- This form is preferable since it involves only first derivatives, is symmetric by construction, and has better variational properties.

`local_function(function=None,cfunction=None):`

- In-place, evaluates function(self(x))
- E.g., f.local_function(g), transforms f() to numerical values on the adaptively refined Legendre quadrature grid, computes g(f(x)) at each grid point, and transforms back.

`mul_func(function=None,cfunction=None):`

- In-place, evaluates pointwise function multiplication
- E.g., f.mul_func(g) transforms f() to numerical values on the adaptively refined Legendre quadrature grid, computes f(x)*g(x) at each grid point, and transforms back.

- These routines are meant as optimizations to avoid first computing compressed representations of g(x)
- Also, they conveniently compute in the support of f(x) (i.e., using the adaptively refined mesh of f(x)) which is often what is desired.
- If autorefine is enabled (see later) the operation is applied after refining one level down.

`scale(alpha):`

- f.scale(a) scales f(x) in place by the scalar a

`gaxpy(alpha, other, beta):`

- f.gaxpy(a,g,b) replaces f(x) by a*f(x)+b*g(x)

- These routines, by operating in place, are intended to be faster and save memory compared with the functionally equivalent Python statements

```
f = a*f
f = a*f + b*g
```

- Note, that if gaxpy() is applied in the scaling function basis, a compress() is eventually required to complete the operation. This is so that multiple gaxpy's may be applied with just one compress.

```
griddx(filename='/tmp/grid.dx'):
plotdx(filename='/tmp/plot.dx',
       npt=(20,20,20),
       box=(0,1,0,1,0,1)):
```

- These two functions write OpenDX files to enable visualization of the grid and the function.
- Have a look at the network in the madness/opendx directory for an example of how to use them.

E.g., Energy of Hydrogen Atom

```
from mra import *
L = 32.0
Function.k = 7
Function.thresh = 1e-5

def psi(x,y,z):
    fac = 102.12922378276677 #sqrt(L**3/pi)
    x,y,z = (x-0.5)*L,(y-0.5)*L,(z-0.5)*L
    return fac*exp(-sqrt(x*x+y*y+z*z))

def V(x,y,z):
    x, y, z = (x-0.5)*L, (y-0.5)*L, (z-0.5)*L
    return -1.0/sqrt(x*x+y*y+z*z)

psi = Function(function=psi)
V   = Function(function=V)
px, py, pz = Del*psi
S  = psi.inner(psi)
VE = (V*psi).inner(psi)/S
TE = 0.5*(px.inner(px)+py.inner(py)+pz.inner(pz))/L**2/S
print "norm",S,"kinetic",TE,"potential",VE,"energy",TE+VE
```

$$\psi(r) = \left(\frac{L^3}{\pi}\right)^{\frac{1}{2}} e^{-r} \qquad S = \langle\psi|\psi\rangle = 1$$

$$T = \langle\psi|-\tfrac{1}{2}\nabla^2|\psi\rangle = \tfrac{1}{2}\langle\nabla\psi|\nabla\psi\rangle$$

$$V = \langle\psi|-r^{-1}|\psi\rangle \qquad E = T + V$$

Output

```
     norm 1.00000000014
  kinetic 0.500000335672
potential -1.00000013916
   energy -0.499999803488
```

- Correct, but numerical compression of the Python functions took 70s. Using C-function is 20x faster since all of the Python loops and function calls are avoided

- Autoswig is provided to make this easy.

Same example using C - I

```
from mra import *
from autoswig import autoswig

L = 32.0
Function.k, Function.thresh = 7, 1e-5

C = autoswig('''
static double L;
double psi(double x, double y, double z) {
  double fac=102.12922378276677; /* sqrt(L*L*L/pi) */
  x=(x-0.5)*L; y=(y-0.5)*L; z=(z-0.5)*L;
  return fac*exp(-sqrt(x*x+y*y+z*z));
}
void *Cpsi = (void *) psi;

double V(double x, double y, double z) {
  x=(x-0.5)*L; y=(y-0.5)*L; z=(z-0.5)*L;
  return -1.0/sqrt(x*x+y*y+z*z);
}
void *CV = (void *) V;
''')

C.cvar.L = L        # Must tell C how big the box is
```

Same Example using C - II

```
# Note use of cfunction rather than function

psi = Function(cfunction=C.cvar.Cpsi)
V   = Function(cfunction=C.cvar.CV)

# Below here, the code is exactly the same as
# in the pure Python example

px, py, pz = Del*psi

S  = psi.inner(psi)
VE = (V*psi).inner(psi)/S
TE = 0.5*(px.inner(px) + py.inner(py) + \
          pz.inner(pz))/L**2/S
print "     norm",S
print "  kinetic",TE
print "potential",VE
print "   energy",(TE+VE)
```

Extending Python with Autoswig

- SWIG (http://www.swig.org)
  - A very powerful tool for interfacing scripting languages such as Python, Perl, Tcl, etc., with C, C++ and even Fortran.
  - Swig can read source code or function prototypes and generate the wrapping code to enable Python to call C functions and to access C global variables.
  - Handling complex types (structures, arrays, classes) is also automated and can be extended though typedefs and typemaps.
  - Autoswig automates running swig, compiling and linking the resulting code to produce a shared library that is loaded into Python as a module.

Autoswig Examples

```
from autoswig import autoswig
# Equivalent Python and C routines
def fred(a,b,c):
    return a*b+c

x = autoswig('int fred(int a,int b,int c){return a*b+c;}')

print fred(3,4,5), x.fred(3,4,5)

# Inplace modification of Python strings (v. bad
# practice since it violates immutability, but
# sometimes useful, e.g., for serializing data)

x = autoswig("void dave(char *a) {a[0] = 'H';}")
a = 'hello'
print a       # ---> “hello”
x.dave(a)
print a       # ---> “Hello”
```

Autoswig – how does it work

- Caches modules in a directory
- Generates a unique integer to label the source code (hash function)
- Tries to load the module
  - If succeeds, then no need to compile
- Writes the source to disk
- Compiles, and links to make a shared lib
- Loads the module
- If any step fails, error message will point you to the relevant log file

Autoswig – overriding defaults

- Can override defaults in the autoswig call, e.g.

```
import autoswig
autoswig.autoswig(code,include_dirs=["/dave/x"],language="C++")
```

- Or the directory of defaults in autoswig module
- Most important things are
  - autoswigdir directory to cache modules
    - default is madness/autoswig/autoswigdir which may not work if you don't have write permission
  - include and library paths
  - have a look in autoswig.py for full list
  - E.g., to modify the actual defaults

```
import autoswig
autoswig.defaults['autoswigdir'] = "……."
autoswig.defaults['language'] = "C++"
```

Multiresolution representation of functions in the multiwavelet basis

- So you've seen how to do it … now for some mathematical detail about what is actually going on
- This is just a thumbnail sketch, but probably enough for you to implement new low-level capabilities

The mathematicians …

Gregory Beylkin
http://amath.colorado.edu/faculty/beylkin/

George I. Fann
gi_fann@pnl.gov

References

- The (multi)wavelet methods in this work are primarily based upon
  - Alpert, Beylkin, Grimes, Vozovoi (SIAM ????)
  - B. Alpert (SIAM Journal on Mathematical Analysis 24, 246-262, 1993).
  - Beylkin, Coifman, Rokhlin (Communications on Pure and Applied Mathematics, 44, 141-183, 1991.)

- The following are useful further reading
  - Daubechies, *"Ten lectures on wavelets"*
  - Walnut, *"An introduction to wavelets"*
  - Meyer, *"Wavelets, algorithms and applications"*
  - Burrus et al, *"Wavelets and Wavelet transforms"*

Why “think” multiresolution?

- It is everywhere in nature/chemistry/physics
  - Core/valence; high/low frequency; short/long range; smooth/non-smooth; atomic/nano/micro/macro scale
- Common to separate just two scales
  - E.g., core orbital heavily contracted, valence flexible
  - More efficient, compact, and numerically stable
- Multiresolution
  - Recursively separate all length/time scales
  - Computationally efficient and numerically stable
  - Coarse-scale models that capture fine-scale detail

How to “think” multiresolution

- Consider a ladder of function spaces

$$V_0 \subset V_1 \subset V_2 \subset \cdots \subset V_n$$

  - E.g., increasing quality atomic basis sets, or finer resolution grids, …
- Telescoping series

$$V_n = V_0 + (V_1 - V_0) + (V_2 - V_1) + \cdots + (V_n - V_{n-1})$$

  - Instead of using the most accurate representation, use the difference between successive approximations
  - Representation on V_0 small/dense; differences sparse
  - Computationally efficient; possible insights

Another Key Component

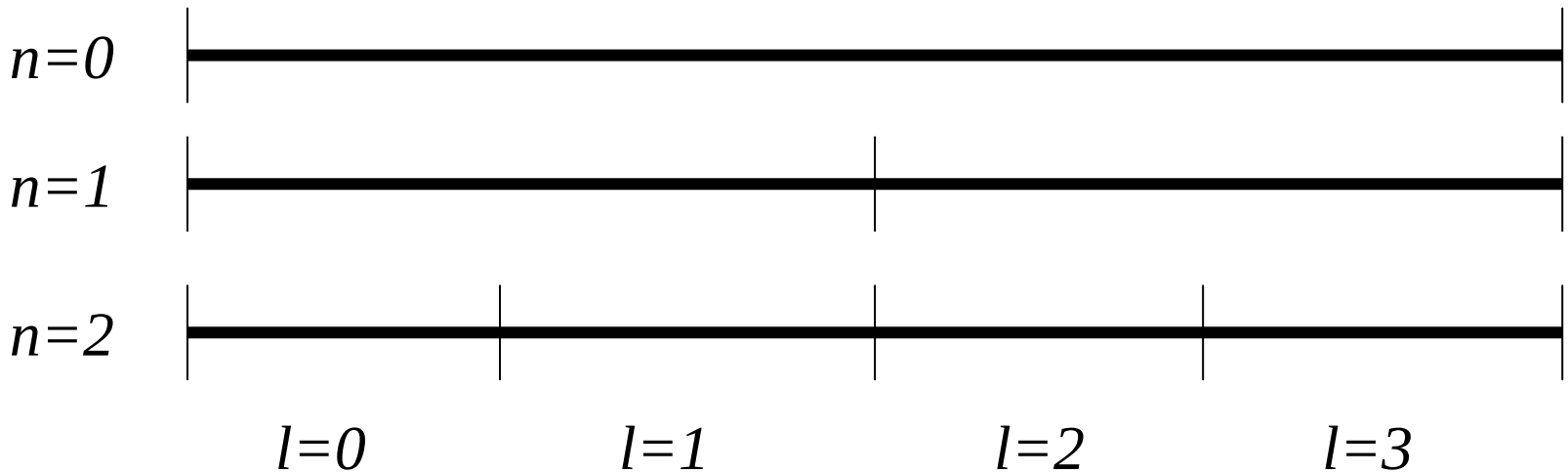
- Trade precision for speed – everywhere
  - Don’t do anything exactly
  - Perform everything to $O(\varepsilon)$
  - Require
    - Robustness
    - Speed, and
    - Guaranteed, arbitrary, *finite* precision

Problem Setup

- In 1-D solve on [0,1]
- Tensor product in n-D
- Periodic and infinite domains also feasible

Scaling Function Basis - I

- Divide domain into 2^n pieces
  - Referred to as level n
  - Adaptive sub-division – local refinement
  - Non-uniform division important (not yet done)
  - l^{th} sub-interval $[l*2^{-n},(l+1)*2^{-n}]$ $l=0,\ldots,n-1$



Scaling Function Basis - II

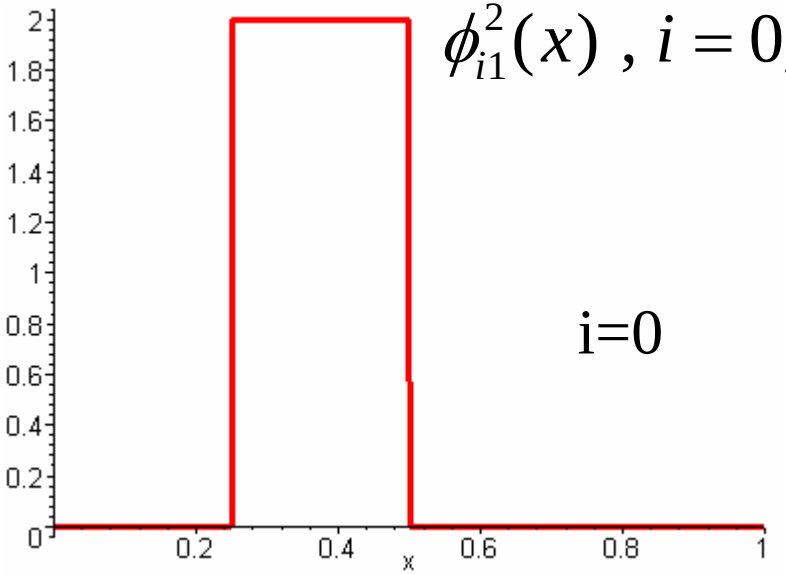
- In each sub-interval define a polynomial basis
  - Currently use the first *k* Legendre polynomials
  - Other bases interesting (e.g, interpolating polynomials, non-polynomial functions)
  - Zero outside of the sub-interval

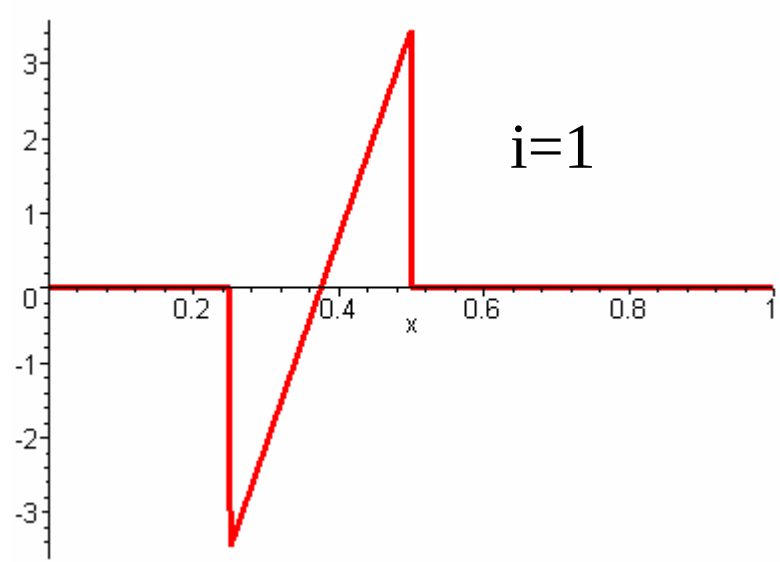
$$\phi_i(x) = \begin{cases} \sqrt{2i+1}P_i(2x-1) & 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

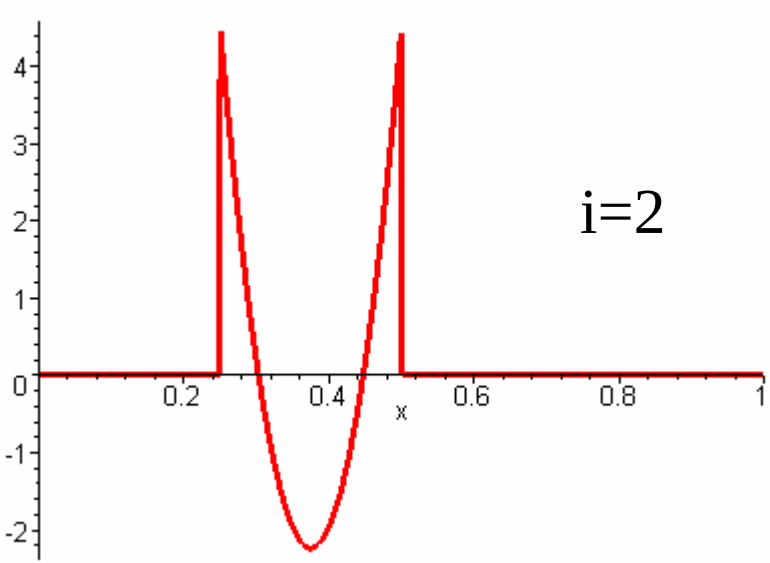
$$\phi_{il}^n(x) = 2^{n/2}\phi_i(2^n x - l)$$

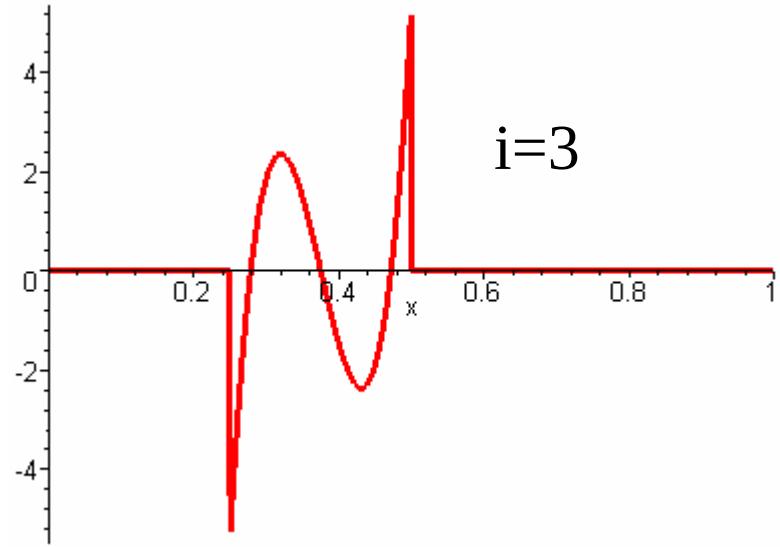
Scaling Function Basis - III

$\phi_{i1}^{2}(x)$, $i = 0,\ldots,3$









Scaling Function Basis - IV

- Translation and dilation $$\phi_{il}^{n}(x) = 2^{n/2}\phi_i(2^n x - l)$$
- Orthonormal $$\int_{-\infty}^{\infty} \phi_{il}^{n}(x)\phi_{jm}^{n}(x) = \delta_{lm}\delta_{ij}$$
- Ladder of spaces $$V_n = \left\{\phi_{il}^{n}(x)\right\}$$ $$i = 0,\ldots,k-1; l = 0,\ldots,2^n - 1$$ $$V_0 \subset V_1 \subset V_2 \subset \cdots$$
- Complete basis for L_2 [0,1] in limit of *either* large *k* or large *n*.

Two-scale relationship

- $V_0 \subset V_1$ – can represent coarse scale basis exactly in the fine scale basis

$$\phi_i(x) = \sqrt{2}\sum_{j=0}^{k-1}\left(h_{ij}^{(0)}\phi_j(2x) + h_{ij}^{(1)}\phi_j(2x-1)\right)$$

$$\phi_{il}^{n}(x) = \sum_{j=0}^{k-1}\left(h_{ij}^{(0)}\phi_{j2l}^{n+1}(x) + h_{ij}^{(1)}\phi_{j2l+1}^{n+1}(x)\right)$$

- The filter coefficients (*H*) may be computed analytically or numerically.

Expansion of a function

- Projection of a function into V_n

$$f^n(x) = \sum_{l=0}^{2^n-1} \sum_{i=0}^{k-1} s_{il}^n \phi_{il}^n(x)$$

$$s_{il}^n = \int dx\, f(x) \phi_{il}^n(x)$$

- Local error is $O(f^{(k)}(\xi)2^{-nk})$ – high-order convergence
- If a function is expanded in V_n, what is the error from truncating to V_{n-1}?
- The error is contained in $W_{n-1}=V_n - V_{n-1}$

Multiwavelet Basis - I

- An orthonormal basis to span $W_{n-1} = V_n - V_{n-1}$
- Also demand
  - Disjoint support (important!)
  - Dilation/translation of wavelets at level 0
- Expand in V_n

$$\psi_i(x) = \sqrt{2}\sum_{j=0}^{k-1}\left(g_{ij}^{(0)}\phi_j(2x) + g_{ij}^{(1)}\phi_j(2x-1)\right)$$

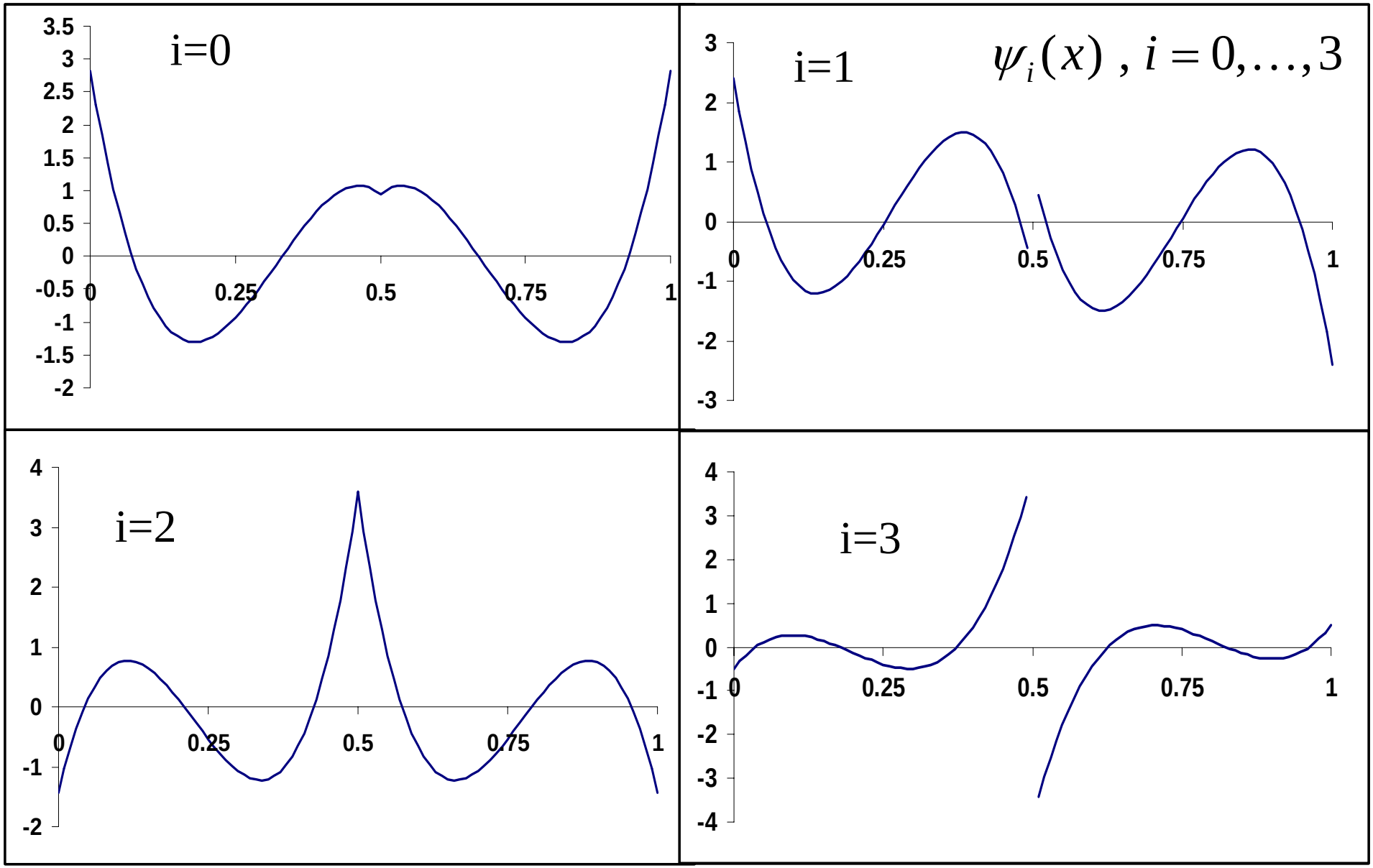
$$\psi_{il}^{n-1}(x) = \sum_{j=0}^{k-1}\left(g_{ij}^{(0)}\phi_{j2l}^{n}(x) + g_{ij}^{(1)}\phi_{j2l+1}^{n}(x)\right)$$

Multiwavelet Basis - II

- Basis not yet completely specified
  - Currently use Alpert's basis (which imposes additional constraints … more about this later)
- The coefficients (*G*) may be evaluated analytically or numerically
  - Numerical evaluation requires extended precision arithmetic (e.g., 208 bits used for k=12).
- Constructive approach
  - From scaling function make the wavelets
  - Often the reverse (e.g., Daubechies wavelets)

Multiwavelet Basis - III

$k=4$



Multiwavelet Basis - IV

- Translation and dilation $$\psi_{li}^{n}(x) = 2^{n/2}\psi_i(2^n x - l)$$
- Orthonormal $$\int_{-\infty}^{\infty} \psi_{li}^{n}(x)\psi_{mj}^{n'}(x) = \delta_{nn'}\delta_{lm}\delta_{ij}$$
- Direct sum of sub-spaces $$V_n = V_0 \oplus W_0 \oplus W_1 \oplus \cdots \oplus W_{n-1}$$
- Complete basis for L_2 [0,1] in limit of *either* large *k* or large *n*.

Multiwavelet Basis - V

- Vanishing moments
  - Critically important property
  - Since W_n is orthogonal to V_n the first k moments of functions in W_n vanish, i.e.,

$$\int x^j \psi_i(x)dx = 0,\ j = 0,\ldots,k-1$$

- In Alpert's basis, additional moments of some of the multiwavelet components also vanish

Some Consequences of Vanishing Moments

- Compact representation of smooth functions
  - Consider Taylor series … the first k terms vanish and smooth implies higher order terms are small
- Compact representation of integral operators
  - E.g., $1/|r-s|$
  - Consider double Taylor series or multipole expansion
  - Interaction between wavelets decays as r^{-2k-1}
- Derivatives at origin vanish in Fourier space
  - Diminishes effect of singularities at that point

Two-scale relationship - I

$$\begin{pmatrix} \phi(x) \\ \psi(x) \end{pmatrix} = \sqrt{2} \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} \begin{pmatrix} \phi(2x) \\ \phi(2x-1) \end{pmatrix}$$

Filter coefficients for k=1 (Haar)

$$\begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$\begin{pmatrix} s_0^0 \\ d_0^0 \end{pmatrix} = \sqrt{2} \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} \begin{pmatrix} s_0^1 \\ s_1^1 \end{pmatrix} = \begin{pmatrix} s_0^1 + s_1^1 \\ s_0^1 - s_1^1 \end{pmatrix}$$

Hence, coefficients of scaling functions and wavelets are often referred to as <u>s</u>um and <u>d</u>ifference coefficients.

Filter coefficients for k=4

7.0711e-01	0.0000e+00	0.0000e+00	0.0000e+00	H0
-6.1237e-01	3.5355e-01	0.0000e+00	0.0000e+00	
0.0000e+00	-6.8465e-01	1.7678e-01	0.0000e+00	
2.3385e-01	4.0505e-01	-5.2291e-01	8.8388e-02	
7.0711e-01	0.0000e+00	0.0000e+00	0.0000e+00	H1
6.1237e-01	3.5355e-01	0.0000e+00	0.0000e+00	
0.0000e+00	6.8465e-01	1.7678e-01	0.0000e+00	
-2.3385e-01	4.0505e-01	5.2291e-01	8.8388e-02	
0.0000e+00	1.5339e-01	5.9409e-01	-3.5147e-01	G0
1.5430e-01	2.6726e-01	1.7252e-01	-6.1237e-01	
0.0000e+00	8.7867e-02	3.4031e-01	6.1357e-01	
2.1565e-01	3.7351e-01	4.4362e-01	3.4233e-01	
0.0000e+00	-1.5339e-01	5.9409e-01	3.5147e-01	G1
-1.5430e-01	2.6726e-01	-1.7252e-01	-6.1237e-01	
0.0000e+00	-8.7867e-02	3.4031e-01	-6.1357e-01	
-2.1565e-01	3.7351e-01	-4.4362e-01	3.4233e-01	

The point being only that these are not mysterious or weird values.

Two-scale relationship II

$$f^n(x) = \sum_{l=0}^{2^n-1} \sum_{i=0}^{k-1} s_{il}^n \phi_{il}^n(x)$$

- May be rewritten without approximation as

$$f^n(x) = \sum_{l=0}^{2^{n-1}-1} \sum_{i=0}^{k-1} \left(s_{il}^{n-1} \phi_{il}^{n-1}(x) + d_{il}^{n-1} \psi_{il}^{n-1}(x) \right)$$

- Where

$$\begin{pmatrix} s_{il}^{n-1} \\ d_{il}^{n-1} \end{pmatrix} = \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} \begin{pmatrix} s_{i2l}^n \\ s_{i2l+1}^n \end{pmatrix}$$

Compression of a function

$$V_n = V_0 \oplus W_0 \oplus W_1 \oplus \cdots \oplus W_{n-1}$$

- Recursively apply the two-scale relation
- The basis is the scaling functions at level 0 and the multiwavelets at all levels

$$f^n(x) = \sum_{i=0}^{k-1} s_{i0}^0 \phi_{i0}^0(x) + \sum_{n'=0}^{n-1} \sum_{l=0}^{2^{n'}-1} \sum_{i=0}^{k-1} d_{il}^{n'} \psi_{il}^{n'}(x)$$

- Haar basis (*k=1*) gives rise to calling coefficients of the scaling functions and wavelets as sums and differences respectively (hence notation *s* and *d*)
- Compression & reconstruction are *O(N)* operations

Two equivalent representations

- Scaling function basis (reconstructed)

$$f^n(x) = \sum_{l=0}^{2^n-1} \sum_{i=0}^{k-1} s_{il}^n \phi_{il}^n(x)$$

- Multi-wavelet basis (compressed)

$$f^n(x) = \sum_{i=0}^{k-1} s_{i0}^0 \phi_{i0}^0(x) + \sum_{n'=0}^{n-1} \sum_{l=0}^{2^{n'}-1} \sum_{i=0}^{k-1} d_{il}^{n'} \psi_{il}^{n'}(x)$$

- Rapid compression/reconstruction
  - Asymptotically faster than the FFT
  - Use most appropriate basis for a given operation

A Third Equivalent Representation

- The function tabulated at the Gauss-Legendre quadrature points in each of the adaptively refined boxes
  - Enables rapid multiplication of functions and application of local functions (e.g., V_{xc})
  - Diagonal transformation from interpolating polynomials

Truncation Error

- To satisfy the global error condition

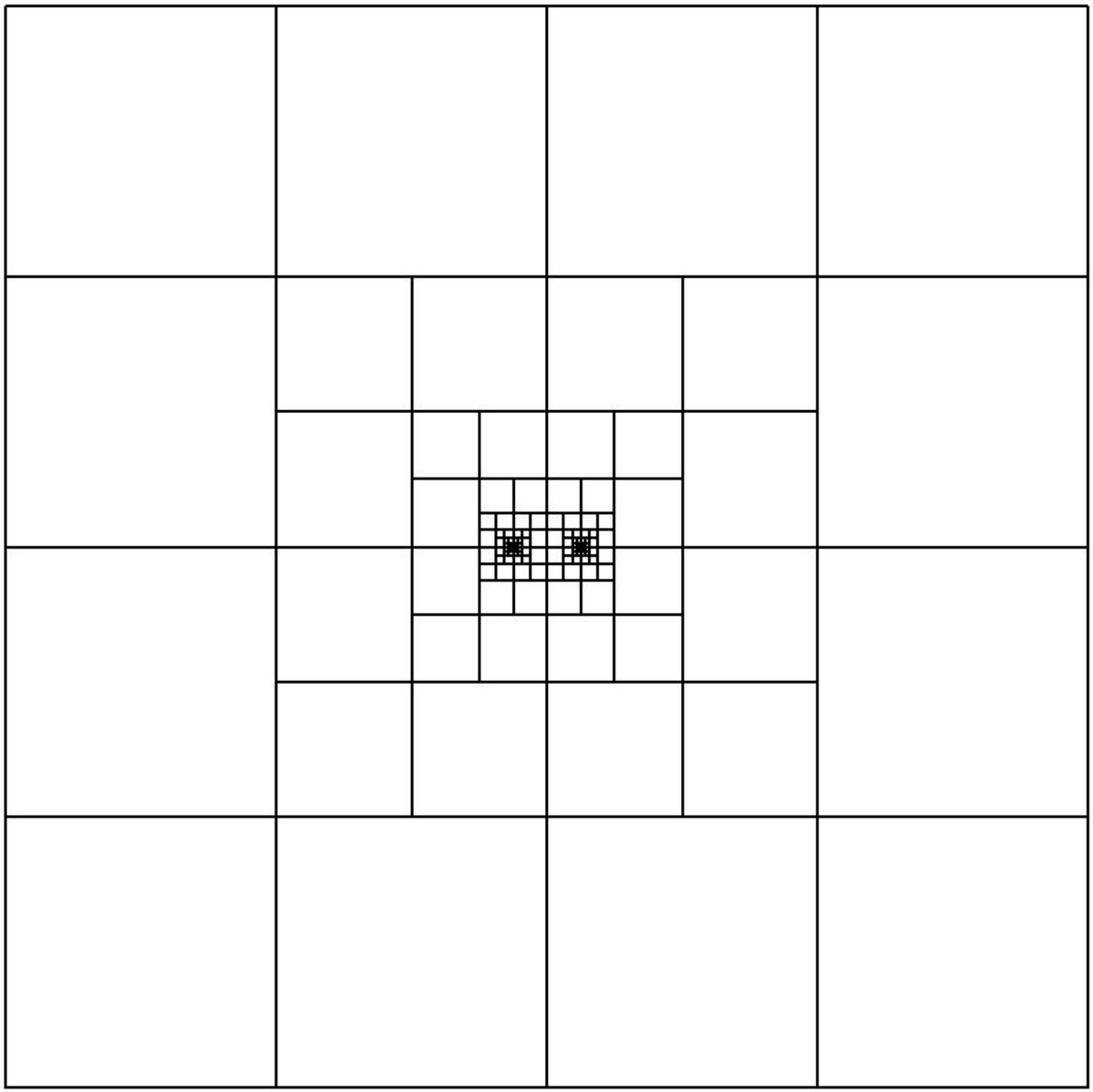
$$\left\| f - f^{n} \right\|_2 \le \varepsilon \left\| f \right\|_2$$

- Truncate according to

$$\left\| d_l^n \right\|_2 \le 2^{-n/2} \varepsilon \left\| f \right\|_2$$

- This is rather conservative – usually use

$$\left\| d_l^n \right\|_2 \le \varepsilon$$



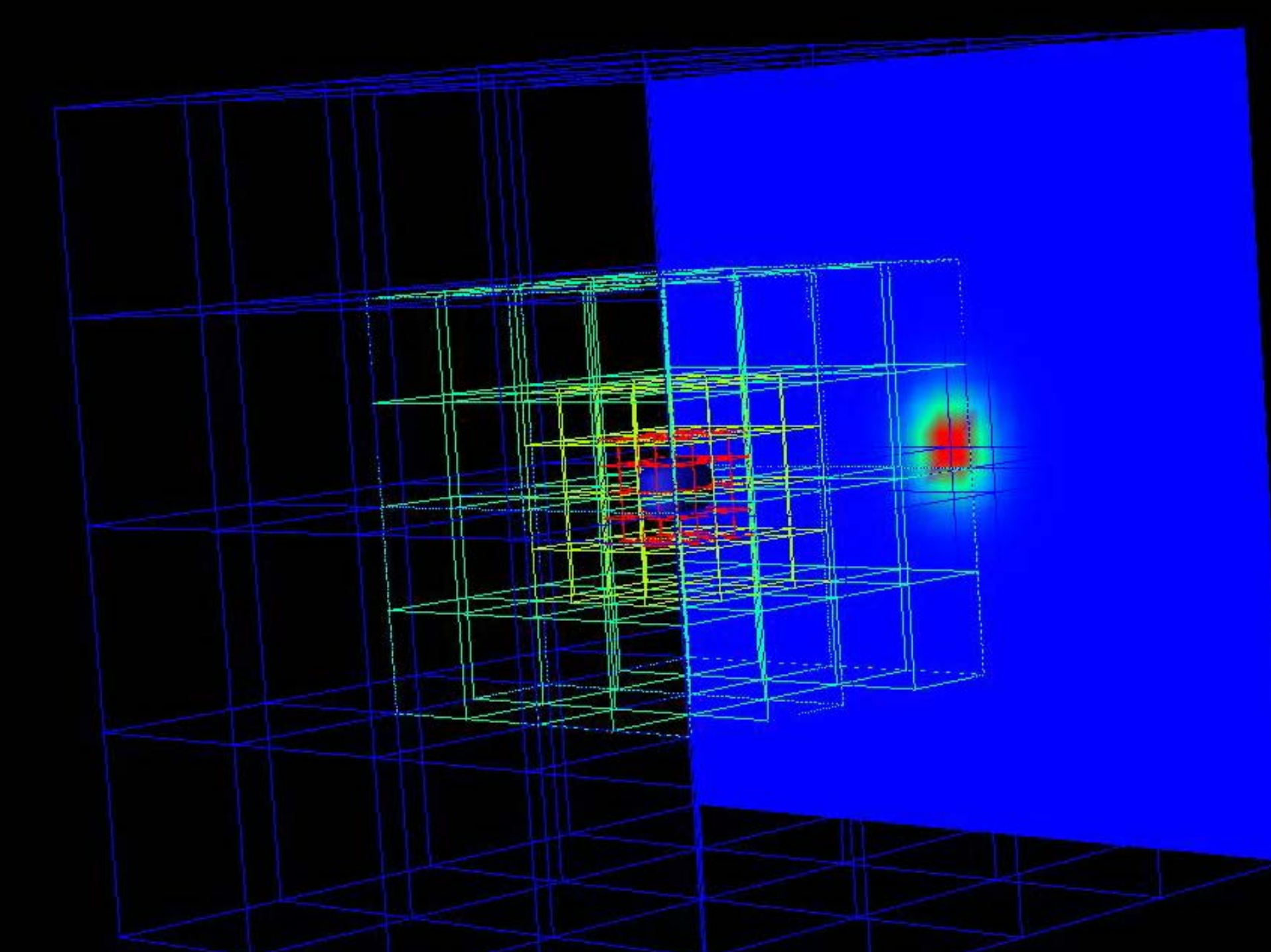
- Slice thru grid used to represent the nuclear potential for H_2 using k=7 to a precision of 10^{-5}.

- Automatically adapts – it does not know a priori where the nuclei are.

- Nuclei at dyadic points on level 5 – refinement stops at level 8

- If were at non-dyadic points refinement continues (to level ??) but the precision is still guaranteed.

- In future will unevenly subdivide boxes to force nuclei to dyadic points.



Summary so far

- Scaling functions
  - Easy to manipulate, evaluate, integrate, …
- Multi-resolution analysis
  - Separates behavior between length scales
  - Local truncation while preserving global error bound
  - Vanishing moments
- Multi-wavelets
  - High-order convergence with adaptive representation
  - Disjoint support – efficient description of singularities if locacated at faces/edges/corners (more efficient than smooth wavelets since they do not have disjoint support)
- Fast compression and reconstruction
  - Orthogonal transformations – numerically stable

Compression of a Matrix

- Compression is just a linear transformation
  - Apply separately to each dimension

$$\tilde{A} = QAQ^T$$

  - Result is said to be in “standard form”

Extension to higher dimensions

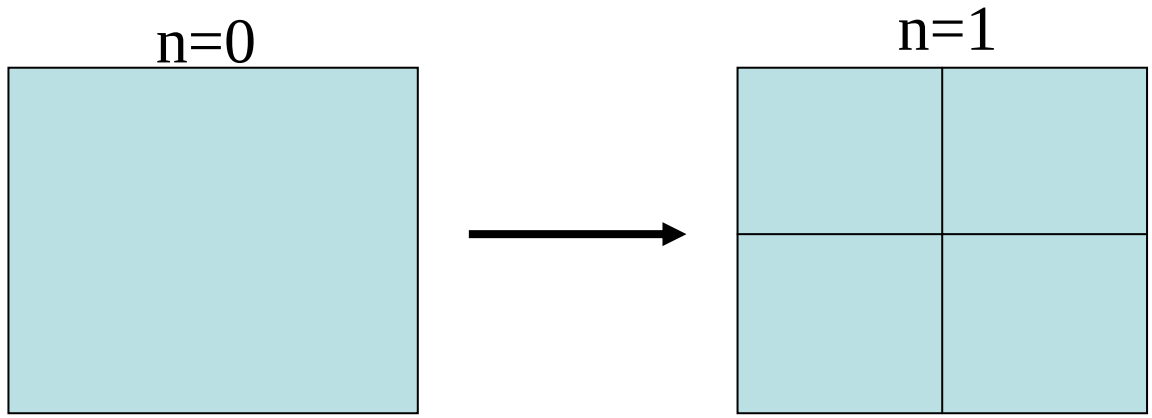
- Scaling function basis is tensor product

$$\phi^n_{ii'll'}(x,y) = \phi^n_{il}(x)\phi^n_{i'l'}(y)$$

- Wavelet basis – tensor product is one choice
  - Standard form – compress each dimension just as for a matrix
  - But cannot refine strictly locally since length scales are mixed between dimensions
- To refine locally need the non-standard form

Non-standard form of functions

- Construct local basis for $W_{n-1}=V_n - V_{n-1}$



$$V_0 = \phi(x)\phi(y)$$

$$V_1 = \sqrt{2}\left\{\begin{array}{l}\phi(2x)\phi(2y), \phi(2x)\phi(2y-1), \phi(2x-1)\phi(2y), \\ \phi(2x-1)\phi(2y-1)\end{array}\right\}$$

$$= \left\{\phi(x)\phi(y), \phi(x)\psi(y), \psi(x)\phi(y), \psi(x)\psi(y)\right\}$$

$$W_0 = V_1 - V_0 = \left\{\phi(x)\psi(y), \psi(x)\phi(y), \psi(x)\psi(y)\right\}$$

Non-standard form of operators - I

- Standard form
  - Matrix elements between different length scales
  - Not very efficient on modern computers with deep memory hierarchies
  - Potentially *O(Nlog N)* non-zero terms
- Non-standard form
  - No matrix elements between lengths scales
  - *O(N)* terms
  - Act on (modified) non-standard form of functions
  - Derivation is instructive

Non-standard form of operators - II

- P_n an orthogonal projection into V_n
- Q_n an orthogonal projection into $W_n = V_{n+1} - V_n$

  $P_n + Q_n = P_{n+1}$
- Consider the projection of an operator T

$$
\begin{aligned}
T_n &= P_n T P_n \\
&= \left(P_{n-1} + Q_{n-1}\right) T \left(P_{n-1} + Q_{n-1}\right) \\
&= P_{n-1} T P_{n-1} + Q_{n-1} T Q_{n-1} + Q_{n-1} T P_{n-1} + P_{n-1} T Q_{n-1} \\
&= T_{n-1} + A_{n-1} + B_{n-1} + C_{n-1} \\
&= T_0 + \sum_{\eta=0}^{n-1} A_\eta + B_\eta + C_\eta
\end{aligned}
$$

NS Form of Operators III

- Matrix elements in the scaling function basis

$$\left[r_{lm}^{n} \right]_{ij} = \int \phi_{il}^{n}(x) \left(T \phi_{jm}^{n}(x) \right) dx$$

- Matrix elements of the NS form

$$\begin{pmatrix} r_{l,m}^{n-1} & \gamma_{l,m}^{n-1} \\ \beta_{l,m}^{n-1} & \alpha_{l,m}^{n-1} \end{pmatrix} = \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} \begin{pmatrix} r_{2l,2m}^{n} & r_{2l,2m+1}^{n} \\ r_{2l+1,2m}^{n} & r_{2l+1,2m+1}^{n} \end{pmatrix} \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix}^{T}$$

Integral operators - I

- Consider $Tf(x) = \mathrm{pv}\int dy K(x-y) f(y)$

$$\left[r_{lm}^{n}\right]_{ij} = \int dx \int dy K(x-y)\phi_{il}^{n}(x)\phi_{jm}^{n}(y)$$

$$= 2^{-n}\int dx \int dy K(2^{-n}\left(x-y+l-m\right))\phi_i(x)\phi_j(y)$$

$$= 2^{-n}\int dz K(2^{-n}\left(z+l-m\right))\Phi_{ij}(z)$$

$$\Phi_{ij}(z) = \int dx \phi_i(x)\phi_j(x-z)$$

$$= \sum_{p=0}^{2k-1}\left(c_{ijp}^{(-)}\phi_p(z+1) + c_{ijp}^{(+)}\phi_p(z)\right)$$

$$\left[r_l^n\right]_p = 2^{-n}\int dz K(2^{-n}\left(z+l\right))\phi_p(z)$$

Integral operators - II

- Matrix elements easy to evaluate from compressed form of kernel $K(x)$
- Application in 1-d is fairly efficient
  - $O(k^2)$ operations
- In 3-d seems to need $O(N_{box}k^6)$ operations
  - Prohibitively expensive
- More intelligent approach
  - $O(N_{box}k^4)$ operations for many “physical” kernels
  - Even better is known to be possible

Vanishing Moments

- Sparse integral operators
  - If the derivatives decay rapidly (i.e., the kernel becomes smoother at long range)
  - See this by Taylor expansion (multipole series)
- Consider NS form of Poisson kernel (*1/r*)

$$\left[\alpha_{lm}^{n}\right]_{ij} = \psi_{il}^{n}(x) \leftrightarrow \psi_{jm}^{n}(y) = O\left(\left|l-m\right|^{-2k-1}\right)$$

$$\left[\beta_{lm}^{n}\right]_{ij} = \psi_{il}^{n}(x) \leftrightarrow \phi_{jm}^{n}(y) = O\left(\left|l-m\right|^{-k-1}\right)$$

$$\left[\gamma_{lm}^{n}\right]_{ij} = \phi_{il}^{n}(x) \leftrightarrow \psi_{jm}^{n}(y) = O\left(\left|l-m\right|^{-k-1}\right)$$

Relationship to the FMM

- Greengard, Rokhlin
- Separate the behavior on different length scales
- Exploit low-rank form of off-diagonal blocks
- Approaches each kernel as a special case
- Highly-optimized, but complex
  - E.g., latest FMM uses seven different representations
- MRA approach is immediately general
  - Simpler than FMM since don’t need to traverse up/down tree
  - Not as fast unless use kernel-specific separated forms

Summary

- Standard and non-standard form of functions and operators
  - Non-standard form facilitates local refinement of functions
  - Non-standard operators easy to compute/apply
- Integral convolution operators
  - Importance of vanishing moments
  - *O(N)* to any finite precision