# MADNESS Implementation Notes

Last Modification: 12/14/2009

For more information please contact:

Robert J. Harrison
Oak Ridge National Laboratory
One Bethel Valley Road
P.O. Box 2008, MS-6367
Oak Ridge, TN 37831

email: harrisonrj@ornl.gov
tel: 865-241-3937
fax: 865-572-0680

# Table of Contents

# Chapter 1

# Implementation Notes

This document provides reference information concerning the mathematics, numerics, algorithms, and design of the multiresolution capabilities of MADNESS. The information herein will be useful to both users of MADNESS and implementers of new capabilities within MADNESS.

## 1.1 ABGV

And references therein: that from which (nearly) all else follows.

B. Alpert, G. Beylkin, D. Gines, L. Vozovoi, Adaptive Solution of Partial Differential Equations in Multiwavelet Bases, *Journal of Computational Physics* **182**, 149-190 (2002).

## 1.2 Legendre scaling functions and multiwavelets

### 1.2.1 Scaling functions

The mother Legendre scaling functions $i = 0, \ldots, k-1$ in 1D are defined as

$$\phi(x) = \begin{cases} \sqrt{2i+1}P(2x-1) & 0 \le x \le 1 \\ 0 & otherwise \end{cases} \tag{1.1}$$

These are orthonormal on $[0, 1]$. The scaling functions scaled to level $n = 0, 1, \ldots$ and translated to box $l = 0, \ldots, 2^n - 1$ span the space $V_n^k$ and are defined by

$$\phi_{il}^n(x) = 2^{n/2}\phi_i(2^n x - 1) \tag{1.2}$$

These are orthonormal on $[2^{-n}l, 2^{-n}(l+1)]$. The scaling functions by construction satisfy the following properties:

- In the limit of either large $k$ or large $n$ the closure of $V_n^k$ is a complete basis for $L_2[0,1]$.
- Containment forming a ladder of spaces $V_0^k \subset V_1^k \subset V_2^k \subset \cdots$.
- Translation and dilation, c.f., (2).
- Orthonormality within a scale $\int_{-\infty}^{\infty} \phi_{il}^n(x)\phi_{jm}^n(x)dx = \delta_{ij}\delta_{lm}$.

The two-scale relationship describes how to expand exactly a polynomial at level $n$ in terms of the polynomials at level *n+1.*

$$\begin{aligned} \phi_i(x) &= \sqrt{2}\sum_{j=0}^{k-1}\left(h_{ij}^{(0)}\phi_j(2x) + h_{ij}^{(1)}\phi_j(2x-1)\right) \\ \phi_{il}^n(x) &= \sum_{j=0}^{k-1}\left(h_{ij}^{(0)}\phi_{j2l}^{n+1}(x) + h_{ij}^{(1)}\phi_{j2l+1}^{n+1}(x)\right) \end{aligned} \tag{1.3}$$

The coefficients $H^{(0)}$ and $H^{(1)}$ are straightforwardly computed by left projection of the first equation in (3) with the fine-scale polynomials.

### 1.2.2 Telescoping series

The main point of multiresolution analysis is to separate the behavior of functions and operators at different length scales. Central to this is the telescoping series which *exactly* represents the basis at level $n$ (the finest scale) in terms of the basis at level zero (the coarsest scale) plus corrections at successively finer scales.

$$V_n^k = V_0^k + \left(V_1^k - V_0^k\right) + \cdots + \left(V_n^k - V_{n-1}^k\right) \tag{1.4}$$

If function is sufficiently smooth in some region of space to be represented at the desired precision at some level, then the differences at finer scales will be negligibly small.

### 1.2.3 Multi-wavelets

The space of wavelets at level $n$ $W_n^k$ is defined as the orthogonal complement of the scaling functions (polynomials) at level *n+1* to those at level *n.* I.e.,

$V_{n+1}^k = V_n^k \oplus W_n^k$. Thus, by definition the functions in $W_n^k$ are orthogonal to the functions in $V_n^k$. The wavelets at level $n$ are constructed by expanding them in the polynomials at level *n+1*

$$\begin{array}{rcl} \psi_i(x) & = & \sqrt{2}\sum_{j=0}^{k-1}\left(g_{ij}^{(0)}\phi_j(2x) + g_{ij}^{(1)}\phi_j(2x-1)\right) \\ \psi_{il}^n(x) & = & \sum_{j=0}^{k-1}\left(g_{ij}^{(0)}\phi_{j2l}^{n+1}(x) + g_{ij}^{(1)}\phi_{j2l+1}^{n+1}(x)\right) \end{array} \tag{1.5}$$

The coefficients $G^{(0)}$ and $G^{(1)}$ are formed by orthogonalizing the wavelets to the polynomials at level n. This determines the wavelets to within a unitary transformation and we follow the additional choices in Alpert's papers/thesis.

The wavelets have these properties

- Decomposition of $V_n^k$

$$V_n^k = V_0^k \oplus W_0^k \oplus W_1^k \oplus \cdots \oplus W_{n-1}^k \tag{1.6}$$

- Translation and dilation $\psi_{il}^n(x) = 2^{n/2}\psi_i(2^n x - l)$
- Orthnormality within and between scales

$$\begin{array}{rcl} \int_{-\infty}^{\infty}\psi_{il}^n(x)\psi_{i'l'}^{n'}(x)dx & = & \delta_{nn'}\delta_{ii'}\delta_{ll'} \\ \int_{-\infty}^{\infty}\psi_{il}^n(x)\phi_{i'l'}^{n'}(x)dx & = & \delta_{ii'}\delta_{ll'} \end{array} \tag{1.7}$$

### 1.2.4 Function approximation in the scaling function basis

A function *f(x)* may be approximated by expansion in the orthonormal scaling function basis at level $n$ with the coefficients obtained by simple projection

$$\begin{array}{rcl} f^n(x) & = & \sum_{l=0}^{2^n-1}\sum_{i=0}^{k=1} s_{il}^n\phi_{il}^n(x) \\ s_{il}^n & = & \int_{-\infty}^{\infty} f(x)\phi_{il}^n(x)dx \end{array} \tag{1.8}$$

The two scale relationships embodied in (3) and (5) may be combined to write the following matrix equation that relates the scaling function basis at one scale with the scaling function and wavelet basis at the next coarsest scale.

$$\begin{array}{rcl} \begin{pmatrix}\phi(x)\\ \psi(x)\end{pmatrix} & = & \sqrt{2}\begin{pmatrix}H^{(0)} & H^{(1)}\\ G^{(0)} & G^{(1)}\end{pmatrix}\begin{pmatrix}\phi(2x)\\ \phi(2x-1)\end{pmatrix} \\ \begin{pmatrix}\phi_l^n(x)\\ \psi_l^n(x)\end{pmatrix} & = & \begin{pmatrix}H^{(0)} & H^{(1)}\\ G^{(0)} & G^{(1)}\end{pmatrix}\begin{pmatrix}\phi_{2l}^{n+1}(x)\\ \phi_{2l+1}^{n+1}(x)\end{pmatrix} \end{array} \tag{1.9}$$

Since the transformation is unitary, we also have

$$\begin{array}{rcl} \begin{pmatrix} \phi(2x) \\ \phi(2x-1) \end{pmatrix} & = & \sqrt{2} \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix}^T \begin{pmatrix} \phi(x) \\ \psi(x) \end{pmatrix} \\ \begin{pmatrix} \phi_{2l}^{n+1}(x) \\ \psi_{2l+1}^{n+1}(x) \end{pmatrix} & = & \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix}^T \begin{pmatrix} \phi_l^n(x) \\ \psi_l^n(x) \end{pmatrix} \end{array} \tag{1.10}$$

In table **??** are the filter coefficients for $k$=4, the only point being that these are plain-old-numbers and not anything mysterious.

Table 1.1: Multi-wavelet filter coefficients for Legendre polynomials, $k = 4$.

| $H^{(0)}$ | | | | $H^{(1)}$ | | | |
|---|---|---|---|---|---|---|---|
| 7.0711e-01 | 0.0000e+00 | 0.0000e+00 | 0.0000e+00 | 7.0711e-01 | 0.0000e+00 | 0.0000e+00 | 0.0 |
| -6.1237e-01 | 3.5355e-01 | 0.0000e+00 | 0.0000e+00 | 6.1237e-01 | 3.5355e-01 | 0.0000e+00 | 0.0 |
| 0.0000e+00 | -6.8465e-01 | 1.7678e-01 | 0.0000e+00 | 0.0000e+00 | 6.8465e-01 | 1.7678e-01 | 0.0 |
| 2.3385e-01 | 4.0505e-01 | -5.2291e-01 | 8.8388e-02 | -2.3385e-01 | 4.0505e-01 | 5.2291e-01 | 8.8 |
| $G^{(0)}$ | | | | $G^{(1)}$ | | | |
| 0.0000e+00 | 1.5339e-01 | 5.9409e-01 | -3.5147e-01 | 0.0000e+00 | -1.5339e-01 | 5.9409e-01 | 3.5 |
| 1.5430e-01 | 2.6726e-01 | 1.7252e-01 | -6.1237e-01 | -1.5430e-01 | 2.6726e-01 | -1.7252e-01 | -6. |
| 0.0000e+00 | 8.7867e-02 | 3.4031e-01 | 6.1357e-01 | 0.0000e+00 | -8.7867e-02 | 3.4031e-01 | -6. |
| 2.1565e-01 | 3.7351e-01 | 4.4362e-01 | 3.4233e-01 | -2.1565e-01 | 3.7351e-01 | -4.4362e-01 | 3.4 |

### 1.2.5 Wavelet transform

The transformation in (10) expands polynomials on level $n$ in terms of polynomials and wavelets on level *n-1*. It may be inserted into the function approximation (8) that is in terms of polynomials at level *n*. This yields an exactly equivalent approximation in terms of polynomials and wavelets on level *n-1*. (I have omitted the multiwavelet index for clarity.).

$$\begin{array}{rcl} f^n(x) & = & \sum_{l=0}^{2^n-1} s_l^n \phi_l^n(x) \\ & = & \sum_{l=0}^{2^{n-1}-1} \begin{pmatrix} s_{2l}^n \\ s_{2l+1}^n \end{pmatrix}^T \begin{pmatrix} \phi_{2l}^n(x) \\ \phi_{2l+1}^n(x) \end{pmatrix} \\ & = & \sum_{l=0}^{2^{n-1}-1} \left( \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} \begin{pmatrix} s_{2l}^n \\ s_{2l+1}^n \end{pmatrix} \right)^T \begin{pmatrix} \phi_l^{n-1}(x) \\ \psi_l^{n-1}(x) \end{pmatrix} \\ & = & \sum_{l=0}^{2^{n-1}-1} \begin{pmatrix} s_l^{n-1}(x) \\ d_l^{n-1} \end{pmatrix}^T \begin{pmatrix} \phi_l^{n-1}(x) \\ \psi_l^{n-1}(x) \end{pmatrix} \end{array} \tag{1.11}$$

The sum and difference (scaling function and wavelet) coefficients at level *n-1* are therefore given by this transformation

$$\begin{pmatrix} s_l^{n-1}(x) \\ d_l^{n-1}(x) \end{pmatrix} = \begin{pmatrix} H^{(0)} & H^{(1)} \\ G^{(0)} & G^{(1)} \end{pmatrix} \begin{pmatrix} s_{2l}^n \\ s_{2l+1}^n \end{pmatrix} \tag{1.12}$$

The transformation may be recursively applied to obtain the following representation of a function in the wavelet basis c.f. (6) with direct analogy to the telescoping series (4).

$$f^n(x) = \sum_{i=0}^{k-1} s_{i0}^0 \phi_{i0}^0(x) + \sum_{n=0,\ldots} \sum_{l=0}^{2^n-1} \sum_{i}^{k-1} d_{il}^n \psi_{il}^n(x) \tag{1.13}$$

The wavelet transformation (13) is unitary and is therefore a very stable numerical operation.

### 1.2.6 Properties of the scaling functions

**Symmetry**

$$\phi_i(x) = (-1)^i \phi_i(1-x) \tag{1.14}$$

**Derivatives**

$$\frac{1}{2\sqrt{2i+1}} \frac{d}{dx} \phi_i(x) = \sqrt{2i-1}\phi_{i-1}(x) + \frac{1}{2\sqrt{2i-5}} \frac{d}{dx} \phi_{i-2}(x) \tag{1.15}$$

**Values at end points**

$$\begin{array}{rcl} \phi_i(0) & = & (-1)^i \sqrt{2i+1} \\ \phi_i(1) & = & \sqrt{2i+1} \\ \frac{d\phi_i}{dx}(0) & = & (-1)^i i(i+1)\sqrt{2i+1} \\ \frac{d\phi_i}{dx}(1) & = & i(i+1)\sqrt{2i+1} \end{array} \tag{1.16}$$

## 1.3 User and simulation coordinates

Internal to the MADNESS implementation, all computations occur in the unit volume in $d$ dimensions $[0,1]^d$. The unit cube is referred to as simulation coordinates. However, the user operates in coordinates that in each dimension $q = 0, \ldots, d-1$ may have different upper and lower bounds $[lo_q, hi_q]$ that represents a diagonal linear transformation between the user and simulation coordinates.

$$\begin{array}{rcl} x_q^{user}(x_q^{\rm sim}) & = & (hi_q - lo_q)\, x_q^{\rm sim} + lo_q \\ x_q^{\rm sim}(x_q^{user}) & = & \frac{x_q^{user} - lo_q}{hi_q - lo_q} \end{array} \tag{1.17}$$

This is a convenience motivated by the number of errors due to users neglecting the factors arising from mapping the user space volume onto the unit cube. More general linear and non-linear transformations must presently be handled by the user.

To clarify further the expected behavior and how/when this mapping of coordinates is performed: All coordinates, values, norms, thresholds, integrals, operators, etc., provided by/to the user are in user coordinates. The advantage of this is that the user does not have to worry about mapping the physical simulation space. E.g., if a user computes the norm of a function what is returned is precisely the value

$$|f|_2^2 = \int_{lo}^{hi} |f(x^{user})|^2 \, dx^{user} \tag{1.18}$$

Similarly, when a user truncates a function with a norm-wise error $\epsilon$ this should be the error in the above norm, and coefficients should be discarded so as to maintain this accuracy independent of the user volume. All sum and difference coefficients, quadrature points and weights, operators, etc. are internally maintained in simulation coordinates. The advantage of this is that the operators can all be consistently formulated just once and we only have to worry about conversions at the user/application interface.

### 1.3.1 Normalization of scaling functions in the user coordinates

The scaling functions as written in equation (2) are normalized in simulation coordinates. Normalizing the functions in user coordinates requires an additional factor of $V^{-1/2}$ where $V$ is the user volume (which is just hi-lo in 1D).

$$\int_{lo}^{hi} \left(V^{-1/2}\phi_{il}^n(x^{\rm sim}(x^{user}))\right)^2 dx^{user} \quad = \quad V^{-1}\int_{lo}^{hi} \phi_{il}^n(x^{\rm sim}(x^{user}))^2 dx^{user} = 1 \tag{1.19}$$

## 1.4 Function approximation

The function is approximated as follows

$$f^n(x^{user}) = V^{-1/2} \sum_{il} s^n_{il} \phi^n_{il}(x^{\rm sim}(x^{user})) \tag{1.20}$$

Note that we have expanded the function in terms of basis functions normalized in the user coordinates. This has several benefits, and in particular eliminates most logic about coordinate conversion factors in truncation thresholds, norms, etc.

### 1.4.1 Evaluation

Evaluation proceeds by mapping the user coordinates into simulation coordinates, recurring down the tree to find the appropriate box of coefficients, evaluating the polynomials, contracting with the coefficients, and scaling by $V^{-1/2}$.

### 1.4.2 Projection into the scaling function basis

The user provides a function/functor that given a point in user coordinates returns the value. Gauss-Legendre quadrature of the same or higher order as the polynomial is used to evaluate the integral

$$\begin{aligned}
s^n_{il} &= V^{-1/2} \int_{lo}^{hi} f(x^{user}) \phi^n_{il}(x^{\rm sim}(x^{user})) dx^{user} \\
&= V^{1/2} \int_0^1 f(x^{user}(x^{\rm sim})) \phi^n_{il}(x^{\rm sim}) dx^{sim} \\
&= 2^{dn/2} V^{1/2} \int_{l2^{-n}}^{(l+1)2^{-n}} f(x^{user}(x^{\rm sim})) \phi_i(2^n x^{\rm sim} - l) dx^{sim} \\
&= 2^{-dn/2} V^{1/2} \int_0^1 f(x^{user}(2^{-n}(x+l))) \phi_i(x) dx \\
&\simeq 2^{-dn/2} V^{1/2} \sum_{\mu=0}^{n_{pt}} \omega_\mu f(x^{user}(2^{-n}(x_\mu + l))) \phi_i(x_\mu)
\end{aligned} \tag{1.21}$$

$x_\mu$ and $\omega_\mu$ are the points and weights for the Gauss-Legendre rule of order $n_{pt}$ over *[0, 1]*.

The above can be regarded as an invertible linear transformation between the scaling function coefficients and the approximate function values at the quadrature points ( $\mu = 0, \ldots, n_{pt}$).

$$\begin{aligned}
s^n_{il} &= 2^{-dn/2} V^{1/2} \textstyle\sum_\mu f_\mu \bar{\phi}_{\mu i} \\
f_\mu &= 2^{dn/2} V^{-1/2} \textstyle\sum_i \phi_{\mu i} s^n_{il}
\end{aligned} \tag{1.22}$$

where

$$\begin{aligned}
f_\mu &= f(x^{user}(2^{-n}(x_\mu + l))) \\
\phi_{\mu i} &= \phi_i(x_\mu) \\
\bar{\phi}_{\mu i} &= \phi_i(x_\mu) \omega_\mu \\
\textstyle\sum_\mu \phi_{\mu i} \bar{\phi}_{\mu i} &= \delta_{ij}
\end{aligned} \tag{1.23}$$

The last line merely restates the orthonormality of the scaling function basis that in the discrete Gauss-Legendre quadrature is exact for the scaling function basis with the choice of the quadrature order $n_{pt} \geq k$.

### 1.4.3 Truncation criteria

Discarding small difference coefficients while maintaining precision is central to speed and drives the adaptive refinement. Different truncation criteria are useful in different contexts.

#### Mode 0 - the default

This truncation is appropriate for most calculations and in particular those that have functions with deep levels of refinement (such as around nuclei in all-electron electronic structure calculations). Difference coefficients of leaf nodes are neglected according to

$$\|d_l^n\|_2 = \sqrt{\sum_i |d_{il}^n|^2} \leq \epsilon \tag{1.24}$$

#### Mode 1

This mode is appropriate when seeking to maintain an accurate representation of both the function and its derivative. Difference coefficients of leaf nodes are neglected according to

$$\|d_l^n\|_2 \leq \epsilon \min(1, L2^{-n}) \tag{1.25}$$

where L is the minimum width of any dimension of the user coordinate volume.

The form for the threshold is motivated by re-expressing the expansion (20) in terms of the mother scaling function and then differentiating (crudely, neglecting continuity with neighboring cells).

$$\begin{array}{rcl} f^n(x^{user}) & = & V^{-1/2} 2^{n/2} \sum_{il} s_{il}^n \phi_i(2^n x^{\mathrm{sim}}(x^{user}) - l) \\ \frac{d}{dx^{user}} f^n(x^{user}) & \simeq & V^{-1/2} 2^{3n/2} (hi - lo)^{-1} \sum_{il} s_{il}^n \phi_i'(2^n x^{\mathrm{sim}}(x^{user}) - l) \end{array} \tag{1.26}$$

Thus, we see that the scale dependent part of the derivative is an extra factor of $2^n$ arising from differentiating the scaling function. We must include the factor hi-lo in order to make the threshold volume independent. Finally, we use the

minimum to ensure that the threshold (25) is everywhere at least as tight as (24).

**Mode 2**

This is appropriate only for smooth functions with a nearly uniform level of refinement in the entire volume. Difference coefficients are neglected according to

$$\|d_l^n\|_2 \leq \epsilon 2^{-nd/2} \tag{1.27}$$

This is the truncation scheme described in ABGV. If this truncation mode discards all difference coefficients at level $n$ it preserves a strong bound on the error between the representations at levels $n$ and $n$ – *1*.

$$\left\|f^n - f^{n-1}\right\|_2^2 = \sum_{l=0}^{2^n-1} \|d_l^n\|_2^2 \leq \sum_{l=0}^{2^n-1} \epsilon^2 2^{-nd} = \epsilon^2 \tag{1.28}$$

However, for non-smooth functions beyond two dimensions this conservative threshold can lead to excessive (even uncontrolled) refinement and is usually not recommended.

### 1.4.4 Adaptive refinement

After projection has been performed in boxes *2l* and *2l+1* at level $n$, the scaling function coefficients may be filtered using  to produce the wavelet coefficients in box $l$ at level *n-1*. If the desired truncation criterion (section **??**) is not satisfied, the process is repeated in the child boxes *4l*, *4l+1*, *4l+2*, *4l+3* at level *n+1*. Otherwise, the computed coefficients are inserted at level $n$.

## 1.5 Unary operations

### 1.5.1 Function norm

Due to the chosen normalization of the scaling function coefficients in (20) both the scaling function and wavelet bases are orthonormal in user-space coordinates, thus

$$\begin{aligned} \|f^n\|_2^2 &= \left\|s_0^0\right\|_2^2 + \textstyle\sum_{m=0}^{n-1} \sum_{l=0}^{2^m-1} \|d_l^m\|_2^2 \\ &= \textstyle\sum_{l=0}^{2^n-1} \|s_l^n\|_2^2 \end{aligned} \tag{1.29}$$

### 1.5.2 Squaring

This is a special case of multiplication; please look below.

### 1.5.3 General unary operation

In-place, point-wise application of a user-provided function (q) to a MRA function (f), i.e., q (f (x)). After optional auto-refinement, the function f (x) is transformed to the quadrature mesh and the function q (f (x)) computed at each point. The values are then transformed back to the scaling function basis. The criterion for auto-refinement is presently the same as used for squaring, but it would be straightforward to make this user-defined.

Need to add discussion of error analysis that can ultimately be used to drive rigorous auto-refinement.

### 1.5.4 Differentiation

ABGV provides a detailed description of the weak formulation of the differentiation operator with inclusion of boundary conditions. There is also a Maple worksheet that works this out in gory detail. We presently only provide a central difference with either periodic or zero-value Dirichlet boundary conditions though we can very easily add more general forms. With a constant level of refinement differentiation takes this block-tri-diagonal form

$$t_{il} = L \sum_j r_{ij}^{(+)} s_{jl-1} + r_{ij}^{(0)} s_{jl} + r_{ij}^{(-)} s_{jl+1} \tag{1.30}$$

where $L$ is the size of the dimension being differentiated. The diagonal block of the operator is full rank whereas the off-diagonal blocks are rank one.

The problems arise from adaptive refinement. We need all three blocks at the lowest common level. The algorithm starts from leaf nodes in the source tree trying to compute the corresponding node in the output. We probe for nodes to the left and right in the direction of differentiation (and enforcing the boundary conditions). There are three possibilities

- Present without coefficients – i.e., the neighbor is more deeply refined. In this instance we loop through children of the target (central) node and invoke the differentiation operation on them, passing any coefficients that we have already found (which must include the central node and the other neighbor due to the nature of the adaptive refinement).

- Present with coefficients – be happy.
- Not present – i.e., the neighbor is less deeply refined. The search for the neighbor recurs up the tree to find the parent that does have coefficients.

Once all three sets of coefficients have been located we will be at the level corresponding to the most deeply refined block. For the other blocks we may have coefficients corresponding to parents in the tree. Thus, we need to project scaling coefficients directly from node $n, l$ to a child node $n', l'$ with $n' \geq n$ and $2^{n'-n}l \leq l' < 2^{n'-n}(l+1)$. Equation (44) tells us how to compute the function value at the quadrature points on the lowest level and we can project onto the lower level basis using (22). Together, these yield

$$s^{n'}_{il'} = 2^{d(n-n')/2} \sum_{\mu} \bar{\phi}_{\mu i} \sum_{j} \phi^{n-n',l,l'}_{\mu j} s^{n}_{jl} \tag{1.31}$$

which is most efficiently executed with the summations performed in the order written.

Recurring down is also a little tricky. We always have at least the coefficients for the central box with translation *l*. This yields children *2l* and *2l+1* which are automatically left/right neighbors of each other.

### 1.5.5 Band-limited, free-particle propagator

The unfiltered real-space kernel of the free-particle propagator for the Schrödinger equation is

$$G_0(x,t) = \frac{1}{\sqrt{2\pi i t}} e^{-\frac{x^2}{2it}} \tag{1.32}$$

For large $x$ this becomes highly oscillatory and impractical to apply exactly. However, when applied to a function that is known to be band limited we can neglect components in $G_0$ outside the band limit which causes it to decay. Furthermore, combining application of the propagator with application of a filter enables us to knowingly control high-frequency numerical noise introduced by truncation of the basis (essential for fast computation) and the high-frequencies inherent in the multiwavelet basis (due both to their polynomial form and discontinuous support).

Explicitly, consider the representation of the propagator in Fourier space

$$\hat{G}_0(k,t) = e^{-i\frac{k^2 t}{2}} \tag{1.33}$$

We multiply this by a filter $F(k/c)$ that smoothly switches near $k = c$ from unity to zero. The best form of this filter is still under investigation, but we presently use a $30^{\text{th}}$-order Butterworth filter.

$$F(k) = \left(1 + k^{30}\right)^{-1} \tag{1.34}$$

For $k \ll 1$ this deviates from unity by about $-k^{30}$. This implies that if frequencies up to a band limit $c_{target}$ are desired to be accurate to a precision $\epsilon$ after $N$ applications of the operator, then we should choose the actual band limit in the filter such that $N(c_{target}/c)^{30} \leq \epsilon$ or $c \geq c_{target}(N/\epsilon)^{1/30}$. For a precision of $10^{-3}$ in the highest frequency (lower frequencies will much more accurate) after $10^5$ steps we would choose $c \geq 1.85 c_{target}$. Similarly, for $k \gg 1$ the filter $F(k)$ differs from zero by circa $k^{-30}$ and therefore we must include in the internal numerical approximation of the operator frequencies about 2x greater than $c$ (more precisely, 2.15x for a precision 1e-10 and 2.5x for a precision of 1e-12).

Specifically, we compute the filtered real-space propagator by numerical quadrature of the Fourier expansion of the filtered kernel. The quadrature is performed over $[-c_{top}, c_{top}]$ where $c_{top} = 2.15 * c$. The wave length associated with a frequency $k$ is $\lambda = 2\pi/k$ and therefore limiting to frequencies less than $c$ implies a smallest grid of $h = \pi/c$. This is oversampled by circa 10x to permit subsequent valuation using cubic interpolation. Finally, the real space kernel is computed by inverse discrete Fourier transform and then cubic interpolation.

Fast and accurate application of this operator is still being investigated. We can apply it either in real space directly to the scaling function coefficients or in wavelet space using non-standard form. Presently, the real-space form is both faster and more accurate.

### 1.5.6 Integral convolutions

This is described in gory detail in ABGV and the first multi-resolution qchem paper but eventually all of that should be reproduced here. For now, we simply take care of the mapping from the user to simulation coordinates and other stuff differing from the initial approach.

We start from a separated representation of the kernel $K(x)$ in user coordinates that is valid over the volume for $x_{lo}^{user} \leq |x^{user}| \leq L\sqrt{(d)}$ to a relative precision $\epsilon^{user}$ (except where the value is exponentially small)

$$K(x^{user}) = \sum_{\mu=1}^{M} \prod_{i=1}^{d} T_i^{(\mu)}(x_i^{user}) + O(\epsilon^{user}) \tag{1.35}$$

Since the error in the approximate is relative it is the same in both user and simulation coordinates.

The most common case is that the kernel is isotropic ( $K(x) = K(|x|)$) and therefore the separated terms do not depend upon direction, i.e., $T_i^{(\mu)} = T^{(\mu)}$ (if it is desired to keep the terms real it may be necessary to treat a negative sign separately). In a cubic box the transformation to simulation coordinates is the same in each dimension and therefore we only need to compute and store each operator once for each dimension. However, in non-cubic boxes the transformation to simulation coordinates is different in each direction making it necessary to compute and store each operator in each direction. Doing this will permit us to treat non-isotropic operators in the same framework, the extreme example of which is a convolution that acts only in one dimension. Presently, this is not supported but it is a straightforward modification.

Focusing now on just one term and direction, the central quantity is the transition matrix element that is needed in user coordinates but must be computed in simulation coordinates

$$\begin{aligned}[r^n_{ll'}]_{ii'} &= L^{-1}\int\int T^{user}(x^{user}-y^{user})\phi^n_{il}(x^{sim}(x^{user}))\phi^n_{i'l'}(y^{sim}(y^{user}))\,dx^{user}\,dy^{user} \\ &= L\int\int T^{user}(L(x^{sim}-y^{sim}))\phi^n_{il}(x^{sim})\phi^n_{i'l'}(y^{sim})\,dx^{sim}\,dy^{sim}\end{aligned} \tag{1.36}$$

where $L$ is the width of the dimension. This enables the identification

$$T^{sim}(x^{sim}) = LT^{user}(Lx^{sim}) \tag{1.37}$$

Internally, the code computes transition matrix elements for $T^{sim}$ in simulation coordinates. If the operator is represented as a sum of Gaussian functions $c^{user}\exp(-\alpha^{user}(x^{user})^2)$ then the corresponding form in simulation coordinates will be $c^{sim} = Lc^{user}$ and $\alpha^{sim} = L^2\alpha^{user}$.

### 1.5.7 Application of the non-standard form

Two things complicate the application of the NS-form operator. The first is specific to the separated representation – we only have this for the actual operator ( $T$) not for the NS-form which is $T^{n+1} - T^n$. Thus at each level we actually apply to the sum and difference coefficients $T^{n+1}$ and subtract off the result of applying $T^n$ to just the sum coefficients. Note that screening must be performed using the norm of $T^n - T^{n-1}$ since it is sparse. Beyond 1D this approach is a negligible computational overhead and the only real concern is possible loss of precision since we are subtracting two large numbers to obtain a small difference. My current opinion is that there is no effective loss of precision since reconstructing the result will produce values of similar magnitude. This is I think a correct argument for the leaf nodes, but the interior nodes might have larger values and hence we could lose relevant digits.

The second issue is what to do about scaling function coefficients at the leaf nodes. Regarding the operator as a matrix being applied to a vector of scaling function coefficients, then the operator is exactly applied by operation upon the sum and difference coefficients at the next level, therefore there is no need to apply the operator to the leaf nodes (this was my initial thinking). However, as pointed out by Beylkin, the operator itself can introduce finer-scale detail which means that we must consider application at the lowest level where the difference coefficients are zero since the operator can introduce difference coefficients at that level.

### 1.5.8 Screening

To screen effectively we need to estimate the norm of the blocks of the non-standard operator and also each term in its separated expansion. We could estimate the largest eigenvalue by using a power method and this is implemented in the code for verification, however, it is too expensive to use routinely, especially for each term in a large separated representation (we would spend more time computing the operator than applying it). Thus, we need a more efficient scheme.

Each term in the separated expansion is applied as

$$R_x R_y \cdots - T_x T_y \cdots \tag{1.38}$$

where $R$ is the full non-standard form of the operator in a given direction which takes on the form

$$R = \begin{pmatrix} T & B \\ C & A \end{pmatrix} \tag{1.39}$$

and $T$ is the block of $R$ that connects sum-coefficients with sum-coefficients. We could compute the Frobenius norm of the operator in (38) simply as $\sqrt{\|R_x\|_F^2 \|R_y\|_F^2 \cdots - \|T_x\|_F^2 \|T_y\|_F^2 \cdots}$ but unfortunately this loses too much precision. Instead, an excellent estimate is provided by

$$\sqrt{\left(\prod_{i=1}^{d} \|R_i\|_F^2\right)\left(\sum_{i=1}^{d} \frac{\|R_i - T_i\|_F^2}{\|R_i\|_F^2}\right)} \tag{1.40}$$

which seems in practice to be an effective upper bound that is made tight (within a factor less than two at least for the Coulomb kernel) by replacing the sum with the maximum term in the sum.

### 1.5.9 Automatic refinement (aka widening the support)

Same as for multiplication ... need explain why this is good.

## 1.6 Binary operations

### 1.6.1 Inner product of two functions

This is conceptually similar to the norm, but since the two functions may have different levels of refinement we can only compute the inner product in the wavelet basis.

$$\int f(x)^* g(x)dx = s_{f0}^{0\ dagger}.s_{g0}^{0} + \sum_{m=0}^{n-1}\sum_{l=0}^{2^m-1} d_{fl}^{m\ dagger}.d_{gl}^{m} \tag{1.41}$$

### 1.6.2 Function addition and subtraction

The most general form is the bilinear operation GAXPY (generalized form of SAXPY) $h(x) = \alpha f(x) + \beta g(x)$ that is implemented in both in-place ($h$ the same as $f$) and out-of-place versions. The operation is implemented in the wavelet basis in which the operation can be applied directly to the coefficients regardless of different levels of adaptive refinement (missing coefficients are treated as zero).

*Need a discussion on screening* – basically if the functions have the same processor map and this operation is followed by a truncation before doing anything expensive, explicit screening does not seem too critical. The need for truncation could be reduced by testing on the size of one of the products (e.g., in a Gramm-Schmidt we know that one of the terms is usually small).

### 1.6.3 Point-wise multiplication

This is performed starting from the scaling function basis. Superficially, we transform each function to values at the quadrature points, multiply the values, and transform back. However, there are three complicating issues.

First, the product cannot be exactly represented in the polynomial basis. The product of two polynomials of order *k-1* produces a polynomial of order *2k-2*. Beylkin makes a nice analogy to the product of two functions sampled on a grid – the product can be exactly formed on a grid with double the resolution. While this is not exact for polynomials it does reduce the error by a factor of $2^{-k}$, where

$k$ is the order of the wavelet. Therefore, we provide the option to automatically refine and form the product at a lower level. This is done by estimating the norm of the part of the product that cannot be exactly represented as follows

$$\begin{array}{rcl} p_l^n & = & \sqrt{\sum_{i=0}^{\lfloor (k-1)/2 \rfloor} \|s_{il}^n\|^2} \\ q_l^n & = & \sqrt{\sum_{i=\lfloor (k-1)/2 \rfloor + 1}^{k-1} \|s_{il}^n\|^2} \\ \epsilon(fg)_l^n & \simeq & p(f)_l^n q(g)_l^n + q(f)_l^n p(g)_l^n + q(f)_l^n q(g)_l^n \end{array} \tag{1.42}$$

Second, the functions may have different levels of adaptive refinement. The two options are to compute the function with coefficients at a coarser-scale directly on the grid required for the finer-scale function, or to refine the function down to same level, which is what we previously choose to do. However, this will leave the tree with scaling function coefficients at multiple levels that must be cleaned up after the operation. Since it essential (for efficient parallel computation) to perform multiple operations at a time on a function, having it in an inconsistent state makes things complicated. If all we wanted to do were perform other multiplication operations, there would be no problem; however this seems to be an unnecessary restriction on the user. It is also faster (2-fold?) to perform the direct evaluation so this is what we choose to do.

Third, the above does not use sparsity or smoothness in the functions and does not compute the result to a finite precision. For instance, if two functions do not overlap their product is zero but the above algorithm will compute at the finest level of the two functions doing a lot of work to evaluate small numbers that will be discarded upon truncation. Eliminating this overhead is crucial for reduced scaling in electronic structure calculations. At some scale we can write each function ($f$ and $g$) in a volume in terms of its usual scaling function approximation at that level ($s$) and the correction/differences ($d$) from *all* finer scales. The error in the product of two such functions is then

$$\epsilon(fg) \simeq \|s_f\| \,.\, \|d_g\| + \|d_f\| \,.\, \|\mathrm{s}_f\| + \|d_f\| \,.\, \|d_g\| \tag{1.43}$$

with a hopefully obvious abuse of notation. Note again that while the scaling function coefficients are as used everywhere else in this document, the difference function ($d$) in (43) is the sum of corrections at *all* finer scales. Thus, by computing the scaling function coefficients at all levels of the tree and summing the norm of the differences up the tree we can compute with controlled precision at coarser levels of the tree. The sum of the norm of differences can also be computed by summing the norm of the scaling function coefficients from the finest level and subtracting the local value.

If many products are being formed, the overheads (compute and memory) of forming the non-standard form are acceptable, but it is desirable to have a less

expensive approach when computing just a few products. The above exploits both locality and smoothness in each function. The main reduction in cost in the electronic structure algorithms will come from locality (finite domain for each orbital with exponential decay outside) and with that in mind we can bound the entire product in some volume using $\|fg\| \leq \|f\| \, . \, \|g\|$. We can compute the norm of each function in each volume by summing the norm of the scaling function coefficients up the tree, which is inexpensive but does require some communication and an implied global synchronization. If in some box the product is predicted to be negligible we can immediately set it to zero, otherwise we must recur down. Since we are not exploiting smoothness, if a product must be formed it will be formed on the finest level.

Thus, we will eventually have three algorithms for point-wise multiplication. Which ones to do first? We answer this question by asking, "what products will the DFT and HF codes be performing?"

- Square each orbital to make the density.
- Multiply the potential against each orbital.
- HF exchange needs each orbital times every other orbital.

The first critical one is potential times orbital. The potential has global extent but the orbitals are localized and we want the cost of each product to be *O(1)* not *O(V)* (*V*, the volume). Similarly, we must reduce the cost of the $O(N^2)$ products in HF exchange to *O(N)*. Therefore, we first did the exact algorithm and will very shortly do one that exploits locality.

**Evaluating the function at the quadrature points**

In (22) is described how to transform between function values and scaling coefficients on the same level. However, for multiplication we will need to evaluate the polynomials at a higher level in a box at a finer level. This is not mathematically challenging but there are enough indices involved that care is necessary. Let $(n, l)$ be the parent box and $(n', l')$ be one of its children, and let $x_\mu$ be a Gauss-Legendre quadrature point in [0,1]. We want to evaluate

$$\begin{aligned} f_\mu &= V^{-1/2} \textstyle\sum_i s^n_{il} \phi^n_{il} \left(2^{-n'} \left(x_\mu + l'\right)\right) \\ &= V^{-1/2} 2^{dn/2} \textstyle\sum_i s^n_{il} \phi_i \left(2^{n-n'} \left(x_\mu + l'\right) - l\right) \\ &= V^{-1/2} 2^{dn/2} \textstyle\sum_i s^n_{il} \phi^{n-n',l,l'}_{\mu i} \end{aligned} \tag{1.44}$$

that has the same form as before but now we must use a different transformation for each dimension due to the dependence on the child box coordinates.

## 1.7 Error estimation

To estimate the error in the numerical representation relative to some known analytic form, i.e.,

$$\epsilon = \|f - f^n\| \tag{1.45}$$

we first ensure we are in the scaling function basis and then in each box with coefficients compute the contribution to $\epsilon$ using a quadrature rule with one more point than that used in the initial function projection. The reason for this is that if $f^n$ resulted from the initial projection then it is exactly represented at the quadrature points and will appear, incorrectly, to have zero error if sampled there. One more point permits the next two powers in the local polynomial expansion to be sampled and also ensures that all of the new sampling points interleave the original points near the center of the box.

## 1.8 Data structures

The d-dimension function is represented as a $2^{\mathrm{d}}$-tree. Nodes in the tree are labeled by an instance of `Key<d>` that wraps the tuple *(n,l)* where $n$ is the level and $l$ is a $d$-vector representing the translation. Nodes are represented by instances of `FunctionNode<T,d>` that presently contains the coefficients and an indicator if this node has children. Nodes without children are sometimes referred to as leaves. Nodes, indexed by keys, are stored in a distributed hash table that is an instance of `WorldContainer<Key<d>,FunctionNode<T,d>>`. This container uses a two-level hash to first map a key to the processes (referred to as the owner) in which the data resides, and then into a local instance of either a GNU `hash_map` or a TR1 `unordered_map`. Since it is always possible to compute the key of a parent, child, or neighbor we never actually store (global) pointers to parents or children. Indeed, a major point of the MADNESS runtime environment is to replace the cumbersome partitioned global address space (global pointer = process id + local pointer) with multiple global name spaces. Each new container provides a new name space. Using names rather than pointers permits the application to be written using domain-specific concepts rather a rigid linear model of memory. Folks familiar with Python will immediately appreciate the value of name spaces.

Data common to all instances of a function of a given dimension ($d$) and data type (`T`, e.g., float, double, float_complex, double_complex) are gathered together into `FunctionCommonData<T,d>` of which one instance is generated per wavelet order ($k$). An instance of the common data is shared read-only between all instances of functions of that data type, dimension and wavelet order. Presently there are some mutable scratch arrays in the common data but these

will be eliminated when we introduce multi-threading. In addition to reducing the memory footprint of the code, sharing the common data greatly speeds the instantiation of new functions which is replicated on every processor.

In order to facilitate shallow copy/assignment semantics and to make empty functions inexpensive to instantiate, a multi-resolution function, which is an instance of `Function<T,d>` contains only a shared pointer to the actual implementation which is an instance of `FunctionImpl<T,d>`. Un-initialized functions (obtained from the default constructor) contain a zero pointer. Only a non-default constructor or assignment actually instantiate the implementation. The main function class forwards nearly all methods to the underlying implementation. The implementation primarily contains a reference to the common data, the distributed container storing the tree, little bits of other state (e.g., a flag indicating the compression status) and a bunch of methods.

Default values for all functions of a given dimension are stored in an instance of FunctionDefaults<d>. These may be modified to change the default values for subsequently created functions. Functions have many options and parameters and thus we need an easy way to specify options and selectively override defaults. Since C++ does not provide named arguments (i.e., arguments with defaults that may be specified in any order rather than relying on position to identify an argument) we adopt the named-parameter idiom. The main constructor for `Function<T,d>` takes an instance of `FunctionFactory<T,d>` as its sole argument. The methods of `FunctionFactory<T,d>` enable setting of options and parameters and each returns a reference to the object to permit chaining of methods. A current problem with `FunctionDefaults` is that it is static data shared between all parallel worlds (MPI sub-communicators). At some point we may need to tie this to the world instance.

Pretty much all memory is reference counted using Boost-like shared pointers. An instance of `SharedPointer<T>`, which wraps a pointer of type `T*`, is (almost) always used to wrap memory obtained from the C++ new operator. The exceptions are where management of the memory is immediately given to a low-level interface. Shared-pointers may be safely copied and used with no fear of using a stale pointer. When the last copy of a shared pointer is destroyed the underlying pointer is freed. With this mode of memory management there is never any need to use the C++ delete operator and most classes do not even need a destructor.

Tensors ...

STOPPED MOST CLEANUP AND WRITING HERE ... more to follow ... sigh

### 1.8.1 Maintaining consistent state of the 2d-tree

The function implementation provides a method `verify_tree()` that attempts to check connectivity and consistency of the compression state, presence of coefficients, child flags, etc, as described below.

A node in the tree is labeled by the key/tuple *(n,l)* and presently stores the coefficients, if any, and a flag indicating if the node has children. In 1D, the keys of children are readily computed as *(n+1,2l)* and *(n+1,2l+1)*. In many dimensions it is most convenient to use the `KeyChildIterator` class. The presence of coefficients is presently determined by looking at the size of the tensor storing the coefficients; size zero means no coefficients.

In the reconstructed form (scaling function basis), a tree has coefficients (a $k^d$ tensor) only at the lowest level. All interior nodes will have no coefficients and will have children. All leaf nodes will have coefficients and will not have children.

In the compressed form (wavelet basis), a tree has coefficients (a $(2k)^d$ tensor) at all levels. The scaling function block of the coefficients is zero except at level zero. Logically, this tree is one level shallower than the reconstructed tree since the scaling function coefficients at the leaves are represented by the difference coefficients on the next coarsest level. However, to simplify the logic in compress and reconstruct and to maintain consistency with the non-standard compressed form (see below), we do not delete the leaf nodes from the reconstructed tree. Thus, the compressed tree has the same set of nodes as the reconstructed tree with all interior nodes having coefficients and children, and all leaf nodes having no coefficients and no children.

In the non-standard compressed form (redundant basis), we keep the scaling function coefficients at all levels and the wavelet coefficients for all interior nodes. Thus, the compressed tree has the same set of nodes as for the other two forms but with all nodes having coefficients (a $(2k)^d$ tensor for interior nodes and a $k^d$ tensor for leaf nodes) and with only leaf nodes having no children.

To keep complexity to a minimum we don't want to introduce special states of the tree or of nodes, thus all operations must by their completion restore the tree to a standard state.

Truncation is applied to a tree in compressed form and discards small coefficients that are logically leaf nodes. Logically, because in the stored tree we still have the empty nodes that used to hold the scaling coefficients. For a node to be eligible for truncation it must have only empty children. Thus, truncation proceeds as follows. We initially recur down the tree and for each node spawn a task that takes as arguments futures indicating if each of its children have coefficients. Leaf nodes, by definition, have no children and no coefficients and immediately return their status. Once a task has all information about the

children it can execute. If any children have coefficients a node cannot truncate and can immediately return its status. Otherwise, it must test the size of its own coefficients. If it decides to truncate, it must clear its own coefficients, delete all of its children, and set its `has _children` flag to false. Finally, it can return its own status.

Adding (subtracting) two functions is performed in the wavelet basis. If the trees have the same support (level of refinement) we only have to add the coefficients. If the trees differ, then in addition to adding the coefficients we must also maintain the has _children flag of the new tree to produce the union of the two input trees. To permit functions with different processor maps to be added efficiently, we loop over local data in one function and send them to nodes of the other for addition. Sending a message to a non-existent node causes it to be created.

## 1.9 Returning new functions – selection of default parameters

When returning a new function there is the question of what parameters (thresholds, distribution, etc.) should be used. There needs to a convention that is consistent with users' intuition as well as mechanisms for forcing different outcomes. We choose to not use `FunctionDefaults`. I.e., `FunctionDefaults` is only used when the user invokes the `Function` constructor to fill unspecified elements of `FunctionFactory`.

### 1.9.1 Unary operations (e.g., scaling, squaring, copying, type conversion)

The result copies all appropriate state from the input.

### 1.9.2 Binary operations (e.g., addition, multiplication)

Writing the binary operation as a C++ method invocation f.op(g) there is a natural asymmetry that for consistency with a unary operation leads to our choice to copy all appropriate state from the leftmost function, i.e., that which method is being invoked.

### 1.9.3 Ternary and higher operations

There are no C++ operators of this form and therefore these will always be of the form `f.op(g,h)` and we make the same choice as made for binary operations.

### 1.9.4 C++ operator overloading and order of evaluation

The main issue with the above convention is clarifying how C++ maps statements with overloaded operators[1] into method/function invocations which includes understanding the order of evaluation[2]. Overloading does not change the precedence or associativity of an operator[3].

Noting * is of higher precedence than + and both are left-to-right associative,

- `f*g+h` becomes `(f*g)+h` becomes `(f.mul(g)).add(h)` and thus the result has the same parameters as `f`.
- `h+f*g` becomes `h+(f*g)` becomes `h.add(f.mul(g))` and thus the result has the same parameters as `h`.
- `f*g*h` has undefined order of evaluation since the two operators have equal precedence, but the compiler is not free to assume that multiplication is commutative and hence the result is either `f.mul(g.mul(h))` or `f.mul(g).mul(h)` which will both inherit the parameters of `f`.

In summary, the result always has the parameters of the leftmost function in any expression. For greatest clarity, introduce parentheses or invoke the actual methods/functions rather than relying upon operator overloading.

### 1.9.5 Overriding above behaviors

Operations that produce results dependent upon thresholds, etc., must provide additional interfaces that permit specification of all controlling parameters which will be used in the operation and preserved in the result. For all other operations, it suffices to make thresholds, etc., settable after the completion of an operation.

---

[1] http://www.difranco.net/cop2334/Outlines/ch18.htm
[2] http://msdn2.microsoft.com/en-us/library/yck2zaey(vs.80).aspx
[3] http://www.difranco.net/cop2334/cpp_op_prec.htm

## 1.10 External storage

I/O remains a huge problem on massively parallel computers and should almost never be used except for checkpoint/restart. Several constraints must be borne in mind. First, we must avoid creating many files since parallel file systems are easily over-whelmed if a few thousand processes simultaneously try to create a few tens of files each. Second, I/O should be performed in large transactions with a tunable number of readers/writers in order to obtain the best bandwidth. Third, we need random access to data so purely serial solutions are not acceptable. Finally, the external data representation should ideally be open and readily accessed by other codes.

For the purposes of I/O, we distinguish two types of objects. First, objects that will be written by a single process with no involvement from other processes. Typically this would be just by the master process and the objects would be small enough to fit into the memory of a single processor. Second, large objects that will be transferred to/from disk in a collective manner with all processes in the world logically participating. Random read and write access must be feasible for both types of objects.

## 1.11 Viewing and editing this document

Under Linux ensure you have the Microsoft true-type fonts installed – they are free. Under Ubuntu install package `msttcorefonts`. Without these the default Linux fonts will cause pagination and other problems, at least with the title pages.

Other than resorting to Latex it does not seem possible to put documents under version control and have the changes merged automatically. Subversion recognizes OpenOffice files as being of mime-type octet-stream and thus treats them as binary, meaning that it does not attempt to merge changing. You must use the OpenOffice compare-and-merge facility to manually merge changes yourself.