npme 1.0

User Manual

Dennis M. Elking

Fielddyne, LLC

delking@fielddyne.com

August 5, 2025

Table of Contents

1. Introduction

npme is a fast code for calculating the potential and its gradient of a collection of charges interacting through a radially symmetric kernel function $f(r)$. Consider *N* charges q_i located at positions $\mathbf{r}_i$. The potential φ_i on particle *i* due to the other charges is given by

$$\varphi_i = \sum_{j \neq i} f(r_{ij}) q_j \tag{1}$$

where $r_{ij} \equiv |\mathbf{r}_i - \mathbf{r}_j|$. npme is an implementation of the non-periodic particle mesh Ewald (NPME) method [1]. NPME is an FFT-based fast particle algorithm with scaling ~O(*N* log *N*) and is based on the (periodic) smooth particle mesh Ewald (PME) method [2]. A brief technical summary of the NPME method is given below in Section 1.1, which is followed by a short description of the npme application and library in Section 1.2.

An outline of this manual is given as follows. Citations for using npme are given in Section 2, and installation instructions are described in Section 3. '**npme**' is the main application for calculating the potential with any of the pre-defined kernels: $f(r) = \frac{1}{r}, r^{\alpha}, \frac{\exp(ik_0 r)}{r}$. A short description of how to use the '**npme**' is given in Section 4. Some useful analysis tools are listed in Section 5. Note that '**npme**' is the main standalone application which may be applied to any of the pre-defined kernel functions. Alternatively, specific kernel function applications are described in Section 6. The specific kernel applications provide simplified examples of using the npme library interface inside other code projects. In addition, users can also define their own radially symmetric kernel functions $f(r)$ and a brief description of how to do this is given Section 7. Additional technical details for the '**npme**' application keywords and library source code are included in Appendices A – D.

1.1 Methodology summary

A key concept in NPME [1] is the kernel function $f(r)$ and its Ewald splitting into short-range $f_s(r)$ and smooth long-range $f_l(r)$ contributions given by

$$f(r) = f_s(r) + f_l(r) \tag{2}$$

Substituting the kernel splitting in eqn. 2 into the potential in eqn. 1 leads to the Ewald expression for potential φ_i given by

$$\varphi_i = \varphi_{i,dir} + \varphi_{i,rec} + \varphi_{i,self} \tag{3}$$

where the direct, reciprocal, and self-terms are defined by

$$\varphi_{i,dir} \equiv \sum_{j \neq i} f_s(r_{ij}) q_j \tag{4}$$

$$\varphi_{i,rec} \equiv \sum_{j} f_l(r_{ij}) q_j \tag{5}$$

$$\varphi_{i,self} \equiv -f_l(0) q_i \tag{6}$$

The smooth long-range kernel $f_l(r)$ is represented by a non-periodic Fourier extension of the form

$$f_l(r) = \sum_{\mathbf{k}} \tilde{f}(\mathbf{k}) \exp\,(i\mathbf{k} \cdot \mathbf{r}) \tag{7}$$

The Fourier extension is substituted into the reciprocal sum in eqn. 5

$$\varphi_{i,rec} \equiv \sum_{\mathbf{k}} \tilde{f}(\mathbf{k}) \exp(i\mathbf{k} \cdot \mathbf{r_i})\, S^*(\mathbf{k}) \tag{8}$$

where

$$S(\mathbf{k}) \equiv \sum_j \exp\left(i\mathbf{k} \cdot \mathbf{r_j}\right) q_j^* \tag{9}$$

By interpolating the complex exponential $\exp(i\boldsymbol{k} \cdot \boldsymbol{r})$ with B-spline polynomials on a grid, the smooth PME method [2] is used to calculate both $S(\mathbf{k})$ and $\varphi_{i,rec}$ with the fast Fourier transform (FFT).

A key feature of NPME [1] is that the Fourier extension of $f_l(r)$ in eqn. 7 is calculated numerically with discrete Fourier transform (DFT) interpolation. By calculating the Fourier extension numerically, complicated Fourier integrals are avoided, which results in:

1) substantial flexibility in the choice of possible kernels $f(r)$ and its Ewald splitting into $f_s(r)$ and $f_l(r)$
2) efficient treatment of anisotropic rectangular volumes

In particular, one can choose a splitting for $f(r) = f_s(r) + f_l(r)$ which optimizes computational performance. A 'derivative matching' (DM) Ewald splitting is introduced in [1] for arbitrary radially symmetric kernels $f(r)$, which also exhibits significant performance properties when compared to conventional Ewald splitting methods.

1.2 Overview of npme

The npme library is written in C++, parallelized with OpenMP, and explicitly vectorized using AVX and AVX-512 intrinsic functions with double precision arithmetic. The current version of npme runs on linux shared memory machines and requires the Intel® compiler. npme includes the following pre-defined radially symmetric kernel functions: $f(r) = \frac{1}{r}, r^{\alpha}, \frac{\exp\,(ik_0 r)}{r}$ where α is real and k_0 is complex. In addition, users may also define

their own radially symmetric kernels for $f(r), f_s(r), f_l(r)$ with C++ classes. See reference [1] for properties that $f_s(r)$ and $f_l(r)$ must satisfy. The npme library also includes optimized functions for calculating exact potential and its gradient using pairwise summation $\sim O(N^2)$.

2. Citation

When using npme, please cite the following references:

[1] D. M. Elking, "A non-periodic particle mesh Ewald method for radially symmetric kernels in free space", Comput. Phys. Comm. **315**, 109739 (2025).
https://doi.org/10.1016/j.cpc.2025.109739

[2] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen, "A smooth particle mesh Ewald method", J. Chem. Phys. **103**, 8577 (1995).

https://doi.org/10.1063/1.470117

3. Installation

npme runs on linux machines and requires the Intel® compiler. **Note**: using non-Intel processors with the Intel® compiler may lead to degraded performance. The latest version of npme (npme v1.2) compiles on the newer Intel® compile, which can now be freely downloaded from:

https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html

Older npme versions (npme v1.0 and npme v1.1) compile with the older Intel® compilers, which required a paid license.

To extract and compile:

```
unzip npme-main.zip
cd npme-main
make -j4
```

npme should compile without issues if the newer Intel® compiler is installed. The first few lines of the makefile specify the compiler name and flags by:

```
# Compiler and flags
CPP       = icpx
BASEFLAGS = -ffast-math -qopenmp -funroll-loops -qmkl -Wno-nan-infinity-disabled -MMD -MP
CFLAGS    = -O3 -march=native $(BASEFLAGS)
```

To run the first example:

```
cd test/01_npme_laplaceDM
./run.sh
```

4. Main Application: npme

The main application **npme** calculates the potential and potential gradient for any of the pre-defined kernel functions:

```
/../npme-main/exe/npme coordCharge.txt keyword.txt
```

See **Appendix A** for file format details and **Appendix B** for a complete list of keywords. npme examples may be found in the following test directories:

```
/../npme-main/test/01_npme_laplaceDM/
/../npme-main/test/02_npme_laplaceOrig/
/../npme-main/test/03_npme_RalphaDM/
/../npme-main/test/04_npme_helmholtzDM/
```

and may be run by executing the following script:

```
./run.sh
```

After executing `./run.sh` for any of the 4 examples, the script will display error metrics and timing information by:

1. generate a random box of 100,000 coordinates/charges
2. calculate the NPME potential for various direct sum cutoffs Rdir
3. calculate the exact potential
4. print the error between NPME and exact potentials
5. print out computational times

5. Additional Tools

- **makeRandomBox**: Generates a box of random charges

```
/../npme-main/exe/makeRandomBox inputFile.txt
```

- **compareV**: Calculates absolute/relative errors between 2 potential files

```
/../npme-main/exe/compare V1.output V2.output
```

6. Specific Kernel Application

The main **npme** application described in Sections 3 and 4 can be used for calculations. However, simplified individual specific kernel applications are also available in:

```
/../npme-main/app/laplaceDM.cpp
/../npme-main/app/RalphaDM.cpp
/../npme-main/app/helmholtzDM.cpp
```

These applications demonstrate how to use the npme library for the pre-defined kernels with minimal setup. The specific kernel applications may be useful to programmers who would like to incorporate the npme library into their own C++ projects. Test examples for the 3 applications may be found in:

```
/../npme-main/test/05_specific_kernel
```

7. User-defined Kernels

The npme source code is located in /../npme-main/src/ under the NPME_Library namespace. The two most important header files include:

- NPME_Interface.h - Top level interface class for npme library
- NPME_KernelFunction.h - Base class definition for real/complex kernel functions

A more detailed description of the source code and header files is given in Appendix C.

An example of a user-defined real kernel function for $f = r$ may be found in:

```
/../npme-main/app/userDefineFunc.cpp
```

An example of applying exact, short-range, and long-range kernel functions to the appropriate top level npme interface class is given in:

```
/../npme-main/app/RalphaDM.cpp
```

An example of constructing kernels with DM Ewald splitting for the $f = \frac{1}{r}$ kernel may be found in:

```
/../npme-main/src/NPME_KernelFunctionLaplace.cpp
```

and is described in Appendix D.

Appendix A: File formats

Note that input (.txt) files with real charges correspond to real kernels and real potentials, while those with complex charges correspond to complex kernels and complex potentials.

1) Input File – Real Charge

```
@Coord
    nCharge 100000
      x1  y1  z1
      x2  y2  z2
      x3  y3  z3
      ..

@ChargeReal
    nCharge  100000
      q1  q2  q3  q4  q5
      q6  q7  q8  q9  q10
      …
```

2) Input File – Complex Charge

```
@Coord
    nCharge 100000
      x1  y1  z1
      x2  y2  z2
      x3  y3  z3
      ..

@ChargeComplex
    nCharge 100000
      qr1  qi1  qr2  qi2  qr3
      qi3  qr4  qi4  qr5  qi5
            …
```

3) Output File – Real Charge

```
@Vreal
  nCharge 100000
      V1  dVdx1  dVdy1  dVdz1
      V2  dVdx2  dVdy2  dVdz2
      V3  dVdx3  dVdy3  dVdz3
            ..
```

4) Output File – Complex Charge

```
@Vcomplex
    nCharge  100000
      Vr1    Vi1     dVdxr1  dVdxi1
      dVdyr1 dVdyi1  dVdzr1  dVdzi1
      Vr2    Vi2     dVdxr2  dVdxi2
      dVdyr2 dVdyi2  dVdzr1  dVdzi2
      Vr3    Vi3     dVdxr3  dVdxi3
      ….
```

5) keyword file example for npme, laplaceDM, RalphaDM, and helmholtzDM apps

```
@Keyword
    FFTmemGB      5.0         (maximum FFT memory in GB)
    vecOption     avx         (none, avx, or avx-512, default = avx)
    BnOrder       6           (B-spline order = even integer, default = 8)
    calcType      pme_exact   ('pme', 'exact', or 'pme_exact', required)
    funcType      Laplace     ('Laplace', 'Helmholtz', or 'Ralpha', required)
    EwaldSplit    DerivMatch ('DerivMatch' or 'LaplaceOrig', req)
    Rdir          7.0         (direct space cutoff, required)
    tol           1.0E-6      (direct space tolerance) (default = 1.0E-6)
    printV        1           (print output potential) (default = 1)
```

Each line contains a ‘keyword’ followed by its ‘value’ separated by white space. Any string data after ‘value’ is ignored. A complete list of keywords is given in Appendix B.

Appendix B: Complete List of Keywords

A complete list of keywords used by the main npme application is given in Table I.

Keyword	Type	Required	Description
funcType	string	yes	'Laplace', 'Helmholtz', or 'Ralpha'
EwaldSplit	string	yes	'DerivMatch' or 'LaplaceOrig'
Rdir	real	yes	direct space cutoff
calcType	string	yes	'pme', 'exact', or 'pme_exact'
nProc	integer	no	number of OpenMP threads
FFTmemGB	real	no	maximum FFT grid memory in gigabytes (default = 10.0)
vecOption	string	no	'none', 'avx', or 'avx-512' (default = 'avx')
BnOrder	integer	no	B-spline order (default = 8)
tol	real	no	precision tolerance (default = 1.0E-6)
printV	bool	no	print potential/potential gradients (default = 1)
printLog	bool	no	print log file (default = 1)
nNeigh	integer	no	number of direct sum cell adjacent neighbors (default = 2)
nCellClust1D	integer	no	number of cells per cluster in 1 dimension, total number of cells per cluster = nCellClust1D^3 (default = 3)
k0_r	real	depends	real component of k_0, required if funcType = 'Helmholtz'
k0_i	real	depends	imaginary component of k_0, required if funcType = 'Helmholtz'
alpha	real	depends	exponent in $f = r^{\alpha}$, required if funcType = 'Ralpha'
N1	integer	no	FFT grid size in x direction, must be a multiple of 4, default = RSEM*
N2	integer	no	FFT grid size in y direction, must be a multiple of 4, default = RSEM*
N3	integer	no	FFT grid size in z direction, must be a multiple of 4, default = RSEM*
n1	integer	no	FFT block size in x direction, N1/2 must be a multiple of n1, default = RSEM*
n2	integer	no	FFT block size in y direction, N2/2 must be a multiple of n2, default = RSEM*
n3	integer	no	FFT block size in z direction, N3/2 must be a multiple of n3, default = RSEM*
nDeriv	integer	no	number of derivatives in DM Ewald splitting, default = RSEM*
a1	real	no	Fourier extension 'a' parameter for *x* direction, default = 4.0
a2	real	no	Fourier extension 'a' parameter for *y* direction, default = 4.0
a3	real	no	Fourier extension 'a' parameter for *z* direction, default = 4.0
del1	real	no	Fourier extension 'del' parameter for *x* direction, default = Rdir
del 2	real	no	Fourier extension 'del' parameter for *y* direction, default = Rdir
del 3	real	no	Fourier extension 'del' parameter for *z* direction, default = Rdir

Table I. List of npme keywords.

- *RSEM (reciprocal sum error model) [1] is a knowledge based error model for automatically finding FFT grid/block sizes and is defined for :
  1) $f(r) = r^{\alpha}$ with $\alpha = -1.0, -2.0, \ldots -7.0$ with BnOrder = 4,6,8,16
  2) $f(r) = \frac{\exp(ik_0 r)}{r}$ with BnOrder = 8,16
- For kernels or BnOrder's outside the above definitions, the user would need to manually specify FFT grid and block sizes

Appendix C: Source Code Description

A high-level description of the npme library contained in /npme-main/src is given as follows. A central concept in the npme library is the kernel function $f(r)$ and its Ewald splitting into short $f_s(r)$ and long $f_l(r)$ range contributions. A kernel function is defined in terms of a C++ class. Base class definitions for real/complex kernel functions are called NPME_KfuncReal/NPME_KfuncComplex and are defined in NPME_KernelFunction.h. Specific examples for exact, short-range, and long-range kernel function implementations may be found in the pre-defined kernel header files: NPME_KernelFunctionLaplace.h, NPME_KernelFunctionHelmholtz.h, and NPME_KernelFunctionRalpha.h corresponding to the $f = \frac{1}{r}, \frac{\exp(ik_0 r)}{r}, r^\alpha$ kernels, respectively. An example of a user-defined kernel function for the $f = r$ kernel is given in /npme-main/app/userDefineFunc.cpp.

Low level direct sum and exact sum routines calculated with the base class kernels NPME_KfuncReal and NPME_KfuncComplex may be found in NPME_PotentialGenFunc.cpp. The input for the low-level direct sum routines is a function class for the short-range kernel function $f_s(r)$. Similarly, the input for the low-level exact sum routines is a function class for the exact kernel function $f(r)$. For the reciprocal sum, the Fourier extension is calculated by evaluating the smooth long-range kernel $f_l(r)$ on a grid. The inputs for the Fourier extension routines are smooth long-range kernels $f_l(r)$ and are contained in NPME_RecSumGrid.cpp. The smooth PME functions require the Fourier extension coefficients stored as an input array and do not directly depend on the smooth long-range kernel.

The direct sum and exact sum for the Laplace $f = \frac{1}{r}$ and Helmholtz $f = \frac{\exp(ik_0 r)}{r}$ kernels may be calculated with the general case kernel routines found in NPME_PotentialGenFunc.cpp. However, there are additional specific direct sum and exact sum routines with inlined Laplace and Helmholtz kernels for some additional performance (~10 – 30%) and are contained in NPME_KernelFunctionLaplace.cpp and NPME_KernelFunctionHelmholtz.cpp. In contrast, there are no inlined direct sum and exact sum routines for the $f = r^\alpha$ kernel. In addition, there are no inlined routines for calculating the Fourier extensions, i.e. all of the routines in NPME_RecSumGrid.cpp use the base class kernel definitions for $f_l(r)$ as input.

NPME_Interface.h contains top level npme library interface classes, which may be used to calculate the NPME and exact potential/potential gradient. The general kernel function interface classes contained in NPME_Interface.h are called NPME_InterfaceReal_GenFunc and NPME_InterfaceComplex_GenFunc. For example, the

setup for NPME_InterfaceReal_GenFunc requires 3 kernel function definitions with base class type NPME_KfuncReal corresponding to the exact, short-range, and smooth long-range kernels. A similar setup holds for the complex case. There are also specific interfaces in NPME_Interface.h for the Laplace $f = \frac{1}{r}$ and Helmholtz $f = \frac{\exp(ik_0 r)}{r}$ kernels, which call the inlined direct sum and exact sum routines described above. As an example, the main interface for the Laplace $f = \frac{1}{r}$ kernel with DM Ewald splitting is called NPME_InterfaceReal_Laplace_DM and its use is illustrated in /npme-main/app/laplaceDM.cpp. An example of setting up the main interface class NPME_InterfaceReal_GenFunc is given in /npme-main/app/RalphaDM.cpp. A brief description of all the source code and header files is given in Table II.

Filename (.cpp and/or .h)	Description
NPME_Interface	Top level interface class for npme library
NPME_Constant	Constants, avx/avx-512 compilation flags, and default values
NPME_EstimateParm	Knowledge based Rec. Sum Error model (RSEM)
NPME_KernelFunction	Base class definition for real/complex kernel functions
NPME_KernelFunctionHelmholtz	Implementation for $f = \frac{\exp(ik_0 r)}{r}$ kernel function
NPME_KernelFunctionLaplace	Implementation for $f = \frac{1}{r}$ kernel function
NPME_KernelFunctionRalpha	Implementation for $f = r^{\alpha}$ kernel function
NPME_FunctionDerivMatch	Contains derivative matching (DM) functions
NPME_PotentialGenFunc	Low level direct sum/exact routines for kernel function class
NPME_PotentialHelmholtz	Low level direct sum/exact routines for inlined Helmholtz kernel
NPME_PotentialLaplace	Low level direct sum/exact routines for inlined Laplace kernel
NPME_RecSumInterface	Interface class for rec. sum
NPME_RecSumGrid	Calculates Fourier extension using smooth kernel function class
NPME_RecSumQ	Calculates smooth PME Q array
NPME_RecSumV	Calculates smooth PME potential/potential gradient
NPME_RecSumSupportFunctions	Support functions for rec. sum
NPME_Bspline	Calculates B-spline by explicit piecewise continuous polynomial
NPME_PartitionBox	Partitions rectangular volume into cells for direct sum
NPME_PartitionEmbeddedBox	Groups cells into clusters for direct sum
NPME_PermuteArray	Permutes coordinate/charge/potential indexes using symmetric group
NPME_AlignedArray	Classes to allocate aligned arrays
NPME_MathFunctions	Various math functions mainly used inside kernel defintions
NPME_VectorIntrinsic	avx and avx-512 transpose and misc. functions
NPME_ReadPrint	Various reading/printing functions
NPME_SupportFunctions	Misc. functions used across the library
NPME_ExtLibrary	Interface functions to MKL library for matrix, FFT, and random number

Table II. Brief description of source code and header files in the npme library.

Appendix D: Kernels with DM Ewald Splitting

User-defined radially symmetric kernel functions for short-range $f_s(r)$ and smooth long-range $f_l(r)$ kernels may use DM Ewald splitting npme library functions if the user also supplies radial derivatives of the form

$$f^p(r) \equiv \left(\frac{1}{r}\frac{d}{dr}\right)^p f(r) \tag{8}$$

for $p = 0,1,..N_{der}$. A simple example of applying DM Ewald splitting to the $f(r) = \frac{1}{r}$ kernel is illustrated in the NPME_Kfunc_Laplace_SR_DM and NPME_Kfunc_Laplace_LR_DM objects defined in /npme-main/src/NPME_KernelFunctionLaplace.h using the real kernel base class definition in /npme-main/src/NPME_KernelFunction.h. A description of NPME_Kfunc_Laplace_SR_DM is given as follows.

D.1 Setting Parameters

The NPME_Kfunc_Laplace_SR_DM object is set with the number of derivative parameter Nder and the direct space cutoff Rdir with the member function ::SetParm(..) as illustrated in Figure 1.

```
bool NPME_Kfunc_Laplace_SR_DM::SetParm (const int Nder,
  const double Rdir, bool PRINT, std::ostream& os)
{
  if (Nder > NPME_MaxDerivMatchOrder)
  {
    std::cout << "Error in NPME_Kfunc_Laplace_SR_DM::SetParm\n";
    char str[2000];
    sprintf(str, "  Nder = %d > %d = NPME_MaxDerivMatchOrder\n", Nder,
      NPME_MaxDerivMatchOrder);
    std::cout << str;
    return false;
  }

  _Nder = Nder;
  _Rdir = Rdir;


  std::vector<double> f(_Nder+1);
  NPME_FunctionDerivMatch_RalphaRadialDeriv (&f[0], _Nder, _Rdir, -1.0);
  if (!NPME_FunctionDerivMatch_CalcEvenSeries (&_a[0], &_b[0],
    &f[0], _Nder, _Rdir))
  {
    std::cout << "Error in NPME_Kfunc_Laplace_SR_DM::SetParm\n";
    std::cout << "NPME_FunctionDerivMatch_CalcEvenSeries failed\n";
    return false;
  }

  if (PRINT)
  {
    ......
  }

  return true;
}
```

Figure 1. Excerpt from /npme-main/src/NPME_KernelFunctionLaplace.cpp for the ::SetParm (..) member function for NPME_Kfunc_Laplace_SR_DM.

Radial derivatives $f^p(r)$ of $f(r) = \frac{1}{r}$ given by

$$f^p(r) = \frac{(-1)^p(2p-1)!!}{r^{2p+1}} \tag{9}$$

are evaluated at r = Rdir with the more general free function NPME_FunctionDerivMatch_RalphaRadialDeriv(..) for calculating radial derivatives of $f(r) = r^\alpha$ with argument $\alpha = -1.0$ and stored in an array f[] of size Nder+1. Next, member variable coefficient arrays _a[] and _b[] of size Nder+1 are calculated with the free function NPME_FunctionDerivMatch_CalcEvenSeries(..) using the radial derivatives, Nder, and Rdir as input. The _a[] coefficients define the short-range kernel by

$$f_s(r) = \begin{cases} f(r) - \sum_{p=0}^{N_{der}} a_p r^{2p} & r \le R_{dir} \\ 0 & r > R_{dir} \end{cases} \tag{10}$$

and the _b[] coefficients define the radial derivative of the short-range kernel by

$$\frac{1}{r}\frac{d}{dr} f_s(r) = \begin{cases} \frac{1}{r}\frac{d}{dr} f(r) - \sum_{p=0}^{N_{der}} b_p r^{2p} & r \le R_{dir} \\ 0 & r > R_{dir} \end{cases} \tag{11}$$

Note the _a[] and _b[] coefficients are related by

$$b_p = 2(p+1)a_{p+1} \tag{12}$$

The first radial derivative $f_s^1 \equiv \frac{1}{r}\frac{d}{dr} f_s(r)$ is needed to calculate gradients of $f(r)$ by

$$\begin{aligned} \frac{\partial f}{\partial x} &= x f_s^1 \\ \frac{\partial f}{\partial y} &= y f_s^1 \\ \frac{\partial f}{\partial z} &= z f_s^1 \end{aligned} \tag{13}$$

D.2 Calculating Kernel and Kernel Gradient with ::Calc(..)

After the parameters are set with ::SetParm(..), the object is ready to use and the ::Print or ::Calc(..) member functions may be called. The implementation of the ::Calc(..) member function which calculates $f_s(r)$ and its gradient is illustrated in Figure 2. Note the sums in eqns. 3 and 4 are calculated with the free function NPME_FunctionDerivMatch_EvenSeriesReal (..). Note there are two implementations of ::Calc(..): 1) one for calculating kernel values only and 2) one for calculating kernel and kernel gradient. In addition, note the same arrays are used for input/output as described in NPME_KernelFunction.h.

```
void NPME_Kfunc_Laplace_SR_DM::Calc (const size_t N,
    double *f0, double *x_fX, double *y_fY, double *z_fZ) const
{
  for (size_t i = 0; i < N; i++)
  {
    const double r2 = x_fX[i]*x_fX[i] + y_fY[i]*y_fY[i] + z_fZ[i]*z_fZ[i];
    const double r  = sqrt(fabs(r2));
    const double r3 = r*r2;

    double f1;
    if (r > _Rdir)
    {
      f0[i] = 0;
      f1    = 0;
    }
    else
    {
      f0[i] = 1.0/r -
              NPME_FunctionDerivMatch_EvenSeriesReal (f1,
                _Nder, &_a[0], &_b[0], r2);
      f1    = -1/r3 - f1;
    }

    x_fX[i]         = x_fX[i]*f1;
    y_fY[i]         = y_fY[i]*f1;
    z_fZ[i]         = z_fZ[i]*f1;
  }
}
```

Figure 2. Excerpt from /npme-main/src/NPME_KernelFunctionLaplace.cpp for the ::Calc (..) member function for NPME_Kfunc_Laplace_SR_DM.

D.3 AVX member functions for ::CalcAVX(..)

The ::CalcAVX(..) member function for the first implementation of calculating $f_s(r)$ but not its gradient is given in Figure 3. *N* must be a multiple of 4 and the input/output arrays must be aligned at 32 byte boundaries. The *x*, *y*, *z* array elements are loaded 4 at-a-time into data types called __m256d, which behaves as an aligned double constant size array of size 4. For example, one can apply a C style pointer type cast to __m256d and print the 4 values out by:

```
__m256d xVec = _mm256_load_pd (&x_f0[count]);
double *xPtr = (double *) xVec;
printf("xPtr = %f %f %f %f\n", xPtr[0], xPtr[1], xPtr[2], xPtr[3]);
```

After the xVec, yVec, zVec are loaded with values, the r2Vec is initialized with _m256_mul_pd (xVec, xVec) which is just the 4 values of xVec [] multiplied with themselves, i.e. the first two elements of r2Vec[] are the values: xPtr[0]*xPtr[0] and xPtr[1]*xPtr[1], respectively. Next, r2Vec[] is updated with a 'fused-multiply-add' (FMA) of yVec with itself and then again with zVec with itself. FMA is a fast method of multiplying two vectors together and then adding the result to a third vector. The sqrt of r2Vec is taken and stored in the output array. Next, 1/r is calculated with NPME_Kfunc_Laplace_AVX and the even

power series is calculated with NPME_FunctionDerivMatch_EvenSeriesReal_AVX. Branching is performed below with the avx intrinsic functions: _mm256_cpm_pd, _mm256_and_pd, and _mm256_andnot_pd functions. Note there are two implementations of both ::CalcAVX(..) and ::CalcAVX512(..). A complete description of all avx and avx-512 vector intrinsic functions is located in: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

```
void NPME_Kfunc_Laplace_SR_DM::CalcAVX (const size_t N,
    double *x_f0, const double *y, const double *z) const
{
  const __m256d zeroVec = _mm256_set1_pd(0.0);
  if (N%4 != 0)
  {
    std::cout << "Error in NPME_Kfunc_Laplace_SR_DM::CalcAVX\n";
    std::cout << "N = " << N << "must be a multiple of 4\n";
    exit(0);
  }

  const size_t nLoop        = N/4;
  const __m256d _RdirVec256 = _mm256_set1_pd( _Rdir );

  size_t count = 0;
  for (size_t i = 0; i < nLoop; i++)
  {
    __m256d xVec, yVec, zVec, r2Vec, rVec;
    xVec  = _mm256_load_pd (&x_f0[count]);
    yVec  = _mm256_load_pd (&y[count]);
    zVec  = _mm256_load_pd (&z[count]);

    r2Vec = _mm256_mul_pd  (xVec, xVec);
    #if NPME_USE_AVX_FMA
    {
      r2Vec  = _mm256_fmadd_pd  (yVec, yVec, r2Vec);
      r2Vec  = _mm256_fmadd_pd  (zVec, zVec, r2Vec);
    }
    #else
    {
      r2Vec  = _mm256_add_pd  (_mm256_mul_pd (yVec, yVec), r2Vec);
      r2Vec  = _mm256_add_pd  (_mm256_mul_pd (zVec, zVec), r2Vec);
    }
    #endif
    rVec  = _mm256_sqrt_pd (r2Vec);

    __m256d f0Vec;
    __m256d f0_AVec;
    __m256d f0_BVec;
    NPME_FunctionDerivMatch_EvenSeriesReal_AVX (f0_AVec, r2Vec, _Nder, &_a[0]);
    NPME_Kfunc_Laplace_AVX (f0_BVec, rVec);

    f0_BVec = _mm256_sub_pd (f0_BVec, f0_AVec);

    //use (f0_BVec) if r < Rdir
    //use (zeroVec) if r > Rdir
    {
      __m256d t0, dless, dmore;
      t0      = _mm256_cmp_pd (rVec, _RdirVec256, 1);

      dless   = _mm256_and_pd    (t0, f0_BVec);
      dmore   = _mm256_andnot_pd (t0, zeroVec);
      f0Vec   = _mm256_add_pd (dless, dmore);
    }

    _mm256_store_pd (&x_f0[count], f0Vec);

    count += 4;
  }
}
```

Figure 3. Excerpt from /npme-main/src/NPME_KernelFunctionLaplace.cpp for the ::CalcAVX (..) member function for NPME_Kfunc_Laplace_SR_DM.