Outline of Pseudo-Spectral Method used in the Sandia/LANL DNS code

Mark Taylor

March 19, 2013

Contents

1 Introduction

The Sandia/LANL DNS code solves viscous fluid dynamics equations in a rectangular domain (2D or 3D). It has many options for the equations (Navier-Stokes, Boussinesq, shallow water) discretization (pseudo spectral, finite differences, hybrid, RK4, RK2, stabilized leapfrog). This document describes the RK4 pseudo-spectral model for modeling isotropic homogeneous turbulence (incompressible Navier-Stokes) in a 3D triply periodic box.

The code is documented in

1. Taylor, Kurien and Eyink, Phys. Rev. E 68, 2003.
2. Kurien and Taylor, Los Alamos Science 29, 2005.

Results from this code have also been used in many additional journal publications. Our largest simulation to date is 4 eddy turnover times of decaying turbulence at 2048^3. We use MPI and allow for arbitrary 3D domain decomposition, (which includes slabs, pencils or cubes0. For a grid of size N^3, it can run on up to $N^3/8$ processors. It has been run on as many as 6144 processors with grids as large as 4096^3.

We solve the equations in a square box of side length 1. The equations are

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{w} \times \mathbf{u} + \nabla p = \nu \Delta \mathbf{u} + \mathbf{f}$$

$$\nabla \cdot \mathbf{u} = 0$$

with vorticity $\mathbf{w} = \nabla \times \mathbf{u}$, forcing function f, and p is the modified pressure given by $p = .5\mathbf{u}^2 + p'$ (with p' being the actual Navier-Stokes pressure). We use a pressure projection method, which determines ∇p so that the solution remains divergence free. In this method, the continuity equation $\nabla \cdot \mathbf{u} = 0$ is eliminated by taking

$$p = \Delta^{-1} \nabla \cdot (-\mathbf{w} \times \mathbf{u} + \mathbf{f})$$

2 Outline of Numerical Method

With the pressure projection method, the Navier-Stokes equations can be written as a single prognostic equation

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{w} \times \mathbf{u} + \nabla \Delta^{-1} \nabla \cdot (\mathbf{w} \times \mathbf{u} - \mathbf{f}) + \nu \Delta \mathbf{u} + \mathbf{f}$$

The terms on the RHS can all be explictly computed via the FFT, and thus the equation can be thought of as a system of ODE's for the Fourier coefficients of $\mathbf{u}$.

2.1 RK4

We use the classic 4th order Runge Kutta scheme with an explicit treatment of diffusion. This requires, for each timestep, 4 evaluations of the right-hand-side (RHS) of the Navier-Stokes equation. All that is required to advance the solution in time is simple linear combinations of these 4 RHS evaluations. So the timestepping procedure (not counting the cost to compute the RHS) is a few loops with no communication at a cost of $O(N^3)$.

2.2 Evaluation of RHS

The RHS of the Navier Stokes equations is

$$-\mathbf{w} \times \mathbf{u} + \nabla \Delta^{-1} \nabla \cdot (\mathbf{w} \times \mathbf{u} - \mathbf{f}) + \nu \Delta \mathbf{u} + \mathbf{f}$$

Denote the Fourier coefficients of $\mathbf{u}$ by $\hat{\mathbf{u}}$, and let $\mathbf{n} = \mathbf{w} \times \mathbf{u}$. We advance in time $\hat{\mathbf{u}}$, so for each of the 4 Runge-Kutta stages we must compute the Fourier coefficients of the RHS. Starting with $\hat{\mathbf{u}}$, the steps are

1. $\mathbf{u} = \text{iFFT}(\hat{\mathbf{u}})$. Cost: 3D inverse FFT
2. $\hat{\mathbf{w}} = \nabla \times \hat{\mathbf{u}}$. (single loop, no communication, cost: $O(N^3)$)

3. $\mathbf{w} = \text{iFFT}(\hat{\mathbf{w}})$. Cost: 3D inverse FFT

4. Compute $\mathbf{n} = \mathbf{w} \times \mathbf{u}$. (single loop, no communication, cost: $O(N^3)$)

5. $\hat{\mathbf{n}}$=FFT($\mathbf{n}$). Cost: 3D FFT

6. Compute $\hat{\mathbf{f}}$. Cost: negligable if forcing implemented in Fourier space.

7. Compute Fourier coefficients of remaining linear terms, $\nabla\Delta^{-1}\nabla\cdot(\hat{\mathbf{n}} - \hat{\mathbf{f}}) + \nu\Delta\hat{\mathbf{u}}$ (two loops, no communication, cost $O(N^3)$).

8. Add intermediate results to form Fourier coefficients of the RHS. Cost: free (can be combined with the previous step).

3 The Parallel FFT

The code uses the data transpose model to implement a parallel FFT. The code has a native data decomposition that can be slabs, pencils or cubes. To implement the parallel FFT, the data is transposed if necessary in order to perform on processor FFTs in each direction. The code can use many different FFT's, but we have found that FFTW is usually the fastest. We use the FFTW version 3 *guru* interface and precompute all the FFTW *plans*.

At present we use our own transpose operator to prepare the data for the on-processor FFTs. It uses non-blocking sends and receives and has the advantage that it can overlap communication with the data re-arrangement necessary to perform stride 1 FFTs. For FFTW, the Georgia tech group has presented results [get ref] showing that for the data-transpose 3D parallel FFT (which means multiple 1D FFTs between each transpose), the best results are obtained if the code repacks the data so that FFTW is only asked to perform stride 1 FFTs.

The transpose operation can also implemented using MPI's MPI_Alltoallv(), called within subcommunicators. We currently do not support this approach, but we have evaluated it on the Cray XT3 using Cray's MPI_alltoallv(). If the MPI_alltoallv() is used, the code must then, after the blocking MPI_alltoallv() has completed, repack the data for stride 1 FFTs. In our tests, the MPI_alltoallv() approach, for large problem sizes was two times *slower* than using the codes internal send/recv transpose routines which effectively hide the cost of the data repacking stride 1 FFTs.

As of this writing, FFTW version 3 does not include a parallel FFT for distributed memory architectures. (They did in version 2). If they re-implement a distributed memory FFT in version 3, it would be good to test this. One problem with using a package parallel FFT is that it makes it difficult to use the vorticity evaluation trick discussed in Section **??**, as this trick certain derivatives to be computed with intermediate products of the parallel FFT.

4 The Parallel FFT (description 2

The code is written in Fortran 90. The only libraries used are MPI and a single-processor real to half-complex FFT. We usually use the open source FFTW package, except on some platforms where the vendor provided single-processor FFT is faster than FFTW. The parallel FFT is implemented with the traditional data-transpose method, meaning that all FFTs are computed on-processor and the data must be transposed between different decompositions to allow for the FFTs to be computed in the x-direction, y-direction and then z-direction. The code allows for arbitrarily sized pencil decompositions in each of the three directions. For example, to compute the x-direction component of the 3D FFT, the $N \times N \times N$ computational domain is distributed amongst the processors so that each processor has the complete column of N data points in the x direction. In this x-pencil decomposition, there are $N \times N$ such columns, which are then equally distributed amongst the processors. To compute the y-direction component of the 3D FFT, this data must be completely re-distributed (via message passing) so that each processor now contains a y-pencil (i.e. a few of the length N columns of data in the y direction). A second transpose from this y-pencil decomposition to a z-pencil decomposition is needed to arrange the data for the z direction component of the FFT, which completes the 3D FFT. Grid space data is thus always stored in an x-pencil decomposition, and after taking the FFT the resulting Fourier coefficients are stored in the z-pencil decomposition. The inverse FFT simply reverses this procedure.

There are several choices in how one can implement this data-transpose operation. If one sets up appropriate sub-communicators, the transpose can be implemented as a single call to MPI_Alltoall() within the sub-communicators. One must then compute the N^2 FFTs of length N, all on-processor, but with a large stride. One can add an additional step and re-arrange the data within each processor to allow for stride 1 FFTs. The stride 1 FFTs are significantly faster, but we have found the increase typically does not offset the cost of re-arranging the data.

To hide the cost of this data re-arranging, we have written a custom implementation of the MPI_Alltoall(), using non-blocking point-to-point MPI messages. Our custom implementation allows us to overlap the interprocessor communication with the on-processor data re-arrangement needed to allow for stride 1 FFTs. The result is a transpose operation which is as fast as the MPI_Alltoall() function, but no further data motion is required to compute stride 1 FFTs. Coupled with FFTW, this procedure results in an extremely efficient parallel 3D FFT.

4.1 MPI_alltoallv() versues Point-to-Point

We last studied this aspect of the parallel FFT in 2006, on a dual core XT3 (Redstorm, before its quad core upgrade). We compared the performance of our DNS code transpose routine (DNS-T) and an MPI_alltoall() based transpose (MPI-T).

We first measured the performance for a mesh of 2048^3 using 1024 processors. We ran this configuration using a slab decomposition, where only the z-direction is distributed among the processors. We then measured the time needed to complete the transpose of the z-direction. DNS-T required 0.41 seconds, while MPI-T was significantly slower, requiring 1.1 seconds, All of our current runs of the DNS code use resolutions and processor counts that allow for a slab decomposition where the DNS-T algorithm has a clear performance advantage over MPI-T.

For the types of very high resolution runs on vary large systems that would be made possible by a DOE INCITE award, we will be required to use a pencil decomposition which requires two types of transposes. We only have one comparison of DNS-T and MPI-T in this regime: a mesh of size 4096^3, using 16384 processors. For our test, each of the 4096 x-y slabs are further divided into 4. The result is each processor works with arrays of size 4096x1024x1. The two types of transposes are the coarsely distributed y-direction (where the transpose consists of few messages of large blocks of data), and the finely distributed z-direction (where the transpose consists of many messages of small blocks of data). For the z-direction transpose (many small messages), DNS-T outperformed MPI-T by a factor of 15 (1.14s verses 17.3s). But in the y-direction (few large messages) MPI-T outperformed DNS-T by a factor of 2.8 (0.68s verses 0.24s). As both transposes are required, this case again shows that DNS-T has a significant advantage over MPI-T. However, the fact that there are regimes where MPI-T is faster show savings can be had by using a hybrid approach. One should also benchmark other size pencil decompositions, for example where each processor works with arrays of size 4096x32x32, where both the y direction and z direction transposes are similar, and have intermediate numbers of messages and message size as compared to the two extreme cases benchmarked above.

5 Complexity

We will ignore the on processor loops, since they require no communication and their total cost is $O(N^3)$, while the cost of the FFT is $O(N^3 \log N)$. Thus the cost of the method is all in the cost of the 3 FFTs. Each FFT is applied to a 3 component vector field, so the total cost is given by 9 scalar 3D FFTs. All the parallel communication is hidden in the FFTs.

The FFT looks something like this. We start with a scalar variable like p, (which could also be one component of the velocity vector $\mathbf{u}$) stored with an x-pencil decomposition:

1. Compute 1D FFT of p (N^2 FFTs in the x-direction)
2. Transpose from x-pencil to y-pencil decomposition
3. Compute 1D FFT of p (N^2 FFTs in the y-direction)
4. Transpose from y-pencil to z-pencil decomposition

5. Compute 1D FFT of p (N^2 FFTs in z-direction)

and end up with $\hat{p}$, stored with a z-pencil decomposition. The Inverse FFT simply reverses the above steps.

To estimate the cost of this algorithm, we just need a model for the transpose operation and N^2 simultaneous 1D FFTs. Many codes first use a real-to-complex FFT, followed by complex-to-complex FFTs for the last two (of size N/2). Our code uses only real-to-real FFTs, always of length N (the cost is identical).

The total cost of doing the 3D FFT and 2 3D iFFTs is thus:

1. 9 N^2 1D real-to-complex FFTs of length N
2. 18 N^2 1D complex-to-real iFFTs of length N
3. 3 x-pencil to y-pencil transposes of size N^3.
4. 3 y-pencil to z-pencil transposes of size N^3.
5. 6 z-pencil to y-pencil transposes of size N^3.
6. 6 y-pencil to x-pencil transposes of size N^3.

5.1 Vorticity Evaluation Trick

There is a trick one can use to eliminate one transpose. In Step 3 above, we only compute the second and third components of $\mathbf{w}$. We use the notation

$$\mathbf{w} = \begin{pmatrix} \mathbf{u}_{3,2} - \mathbf{u}_{2,3} \\ \mathbf{u}_{1,3} - \mathbf{u}_{3,1} \\ \mathbf{u}_{2,1} - \mathbf{u}_{1,2} \end{pmatrix}$$

where $\mathbf{u}_{i,j}$ is the j'th derivative of the i'th component. A small savings can be had if the third component of the vorticity, $\mathbf{w}_3$ is instead computed during the step when $\mathbf{u}$ is computed via the iFFT from $\hat{\mathbf{u}}$. While computing the inverse FFT of $\mathbf{u}$, one can also efficiently compute $\mathbf{u}_{2,1}$ and $\mathbf{u}_{1,2}$. This requires the same number of 1D FFTs and y-to-x pencil transposes as would be required to compute $\mathbf{w}_3$, but it has the advantage of requiring one less z-to-y pencil transpose. With a slab decompostion, this will reduce the amount of communication by 1/9. With a pencil decompostion, the reduction is 1/18.

5.2 Other Savings

I know of one other trick which can be used to recover the vorticity in grid space from the intermediate products of the $\mathbf{u} = \text{iFFT}(\hat{\mathbf{u}})$ operation. It will reduce the number of 1D FFTs from 27 to 24, but requires doing each component of the $\mathbf{u} = \text{iFFT}(\hat{\mathbf{u}})$ operation in a different order. (i.e. for one component we need to do the first the iFFT in the z-direction, then the y-direction and then the x-direction, but for another component we must first do the x-direction, then the y-direction and then the z-direction.) This will introduce additional distributed transposes which I conjecture will make the algorithm less efficient for large processor counts.

And of course there could be many other tricks I dont know about!

5.3 Code Specific Transpose Details

Our code actually allows for a full 3D data decomposition (not that useful for the full pseudo spectral code, but useful for some finite difference/spectral hybrids). Because of this, for some configurations the FFT involves two extra steps. The code is defined to have a *reference* decomposition which can be slabs, pencils or cubes.

In the optimal case, if one is careful to choose an reference decomposition which contains complete y-pencils with appropriate padding (no padding in x direction, end padding of 2 in y-direction and no front padding in z direction), the FFT algorithm (in the code, see the subroutines `zx_ifft3d()` and `zx_fft3d()` in `fftops.F90`) looks like this:

1. Compute 1D FFT of p (N^2 FFTs in the x-direction, on processor)
2. Transpose from x-pencil to y-pencil decomposition
3. Compute 1D FFT of p (N^2 FFTs in the y-direction, on processor)
4. Transpose from y-pencil to z-pencil decomposition
5. Compute 1D FFT of p (N^2 FFTs in z-direction, on processor)

If the reference decomposition happens to be x-y slabs, then the first transpose (step 2 above) involves no communication.

If the reference decomposition is not set up for the optimal case above, then the FFT algorithm reverts to this:

1. Compute 1D FFT of p (N^2 FFTs in the x-direction, on processor)

2. Transpose from x-pencil to reference decomposition
3. Transpose from reference to y-pencil decomposition
4. Compute 1D FFT of p (N^2 FFTs in the y-direction, on processor)
5. Transpose from y-pencil to reference decomposition
6. Transpose from reference to z-pencil decomposition
7. Compute 1D FFT of p (N^2 FFTs in z-direction, on processor)

This algorithm contains two extra steps. For this less optimal case, the most efficient configuration is if the reference decomposition happens to be x-y slabs. In that case, then the two extra steps (and the x-pencil transposes) are all done on-processor (simple memory copies). For performance at high resolution, one is usually required to use y-pencils to allow for more parallization. If the reference decomposition is y-pencils, then again the two extra steps are on-processor memory copies with only a small cost. They also serve to arrange the data so all FFTs are done with a stride of 1 and thus their cost is offset by allowing for a more efficient FFT.

Thus with our code, for efficiency, it is best to use a reference decomposition of y-pencils (requiring two distributed tranposes for each FFT) or x-y slabs (requiring one distributed transpose for each FFT). Using x-pencils or z-pencils for the reference decomposition is very inefficient.

5.4 Cost of a single transpose

Take a computational grid of size N^3, and assuming the code is using a reference decompostion of y-pencils, so the domain decompostion looks like $N_1 \times 1 \times N_2$, with the total number of processes $N_p = N_1 N_2$. Then the distributed tranposes are just the x-pencil to/from y-pencil and y-pencil to/from z-pencil routines.

For the x-pencil to/from y-pencil transpose the cost is

1. MPI_Isend: $N_1 - 1$ messages of size $N^3/(N_1^2 N_2)$
2. MPI_Irecv: $N_1 - 1$ messages of size $N^3/(N_1^2 N_2)$

For the y-pencil to/from z-pencil transpose the cost is

1. MPI_Isend: $N_2 - 1$ messages of size $N^3/(N_2^2 N_1)$

2. MPI_Irecv: $N_2 - 1$ messages of size $N^3/(N_2^2 N_1)$

Each scalar tranpose requires the network transmit a total just shy of $2N^3$ real*8 numbers.

5.5 Total cost of the code

The total cost of the code, for 1 Runge-Kutta stage, in terms of communication and flops, and assuming a y-pencil reference decomposition (with the total number of processes $N_p = N_1 N_2$) and a grid of N^3 can thus be estimated as

1. $9N^2$ 1D real-to-complex FFTs, $18N^2$ 1D complex-to-real FFTs. (length N)
2. Each process: MPI_Isend: $9(N_1 - 1)$ messages of size $8N^3/(N_1^2 N_2)$ bytes.
3. Each process: MPI_Irecv: $9(N_1 - 1)$ messages of size $8N^3/(N_1^2 N_2)$ bytes.
4. Each process: MPI_Isend: $8(N_2 - 1)$ messages of size $8N^3/(N_2^2 N_1)$ bytes.
5. Each process: MPI_Irecv: $8(N_2 - 1)$ messages of size $8N^3/(N_2^2 N_1)$ bytes.

Note: The 9 which appears above comes from combining the messages from the (3) x-pencil-to-y-pencil and the (6) y-pencil-to-x-pencil. The 8 comes from combining the messages from the (3) y-pencil-to-z-pencil and the (5) z-pencil-to-y-pencil transposes.

For the total ammount of bytes that must be sent over the network, add the above numbers and multiply by the number of processes to get: $34 \cdot 8N^3$ bytes. (When using a slab decomposition, this is $16 \cdot 8N^3$.) These numbers are per Runge-Kutta stage. One timestep requires 4 stages, so for the total per timestep we need to multiply by 4.

I've also measured the FLOP count per timestep with the following procedure: Using a hardware counter, measure the FLOP count for a run with 5 timesteps and 4 timesteps, and take their difference, to get the FLOP count per timestep. I did this using the FFT91 Fourier Transform, with resolutions N=16, 32, 64,128 and 256. Curve fitting the data resultant data gave the following:

FLOP count estimate per timestep for N^3 grid, N a power of 2:

$$4(297N^3 + (27)2.28N^3 \log_2(N))$$

Repeating this procedure with resolutions of N=12, 24, 48, 96 and 192, gave

FLOP count estimate per timestep for N^3 grid, N a power of 2 and one power of 3:

$$4(340N^3 + (27)2.24N^3 \log_2(N))$$

Since there are 4 Runge-Kutta stage per timestep, the quantity in brackets can be considered the cost for each stage. These estimates, for the ranges of N given, were accurate to better than 1%.

Note: the asymptotic cost of a single FFTs is $2.5N \log_2(N)$. The actual cost of a particular implementation can only be higher than this. So the curve fitted results above (which suggest $2.25N \log_2(N)$), while accurate for $N \sim 256$, probably need some adjustmet for large N.

6 Memory requirements

The storage requirements of the code, for a grid of size N^3 are approximatly 18 real*8 arrays of size N^3, for a total of $144N^3$ bytes.

The memory usage is fully scalable. If there are N_p processes, each one requires no more than $144N^3/N_p$ bytes.

7 Dealiasing

On a computational grid of size N^3, the Fourier coefficient $\hat{\mathbf{u}}(l, m, n)$ has a wave number given by $(2\pi/L)(l, m, n)$, where L is the length of the domain.

$$\Delta x = \frac{L}{N} \qquad k_{\max} = \frac{2\pi}{L}\frac{N}{2} = \frac{\pi}{\Delta x}$$

In our code, $L = 1$, and thus the wave numbers are given as integer multiples of 2π. Many codes use a domain of side length length $L = 2\pi$, in which case the wave numbers are integers. When comparing parameters from different runs, it is important to recognize that all *dimensional* values (length scales, energies, dissipation rates) depend on L.

Dealiasing is necessary when computing the nonlinear term $\mathbf{w} \times \mathbf{u}$. Dealiasing will remove some of the high wave numbers, so in practice the effective $k_{\max}$ will depend on the type of dealiasing used. There are many types of dealiasing that can be used. The types supported in the Sandia/LANL DNS code include:

1. 2/3 rule (DNS code option: fft-dealais)

2. Phase shifting plus spherical truncaton (DNS code option: fft-phase)

3. Spherical truncation (only partial dealiasing, DNS code option: fft-sphere)

7.1 Dealiasing Recommendations

For many problems, exact dealiasing is not necessary and then simple spherical truncation is recommended as it is by far the most efficient.

When exact dealiasing is needed, then Phase shifting plus spherical truncation is recommended, as it is the most efficient in terms of flops per fully retained retained spherical wave number shells. Note that phase shifting plus spherical truncation requires exactly twice as many FFTs and transposes as simple spherical truncation. (So the complexity analysis in the previous section needs to be adjusted accordingly).

The 2/3 rule is also sometimes usefull. For a given grid, it requires no additional FFTs. It is most efficient in terms of flops per retained wave numbers, but many of the retained wave numbers are not that usefull as they lie outside the largest fully retained spherical shell of wave numbers retained by the truncation.

7.2 2/3 rule

The 2/3 rule sets to zero all coefficients all wave numbers larger than $(2\pi)(2/3)(N/2)$. To be precise, we set to zero (or do not even compute)

$$\hat{\mathbf{u}}(l,m,n) = 0 \qquad \text{for } l > \frac{N}{3}, \quad m > \frac{N}{3}, \text{ or } n > \frac{N}{3}.$$

With this type of dealiasing, we define the maximum wave number in our simulation to be $k_{\max} = 2\pi N/3$. The total number of modes retained is proportional to $(2N/3)^3$.

7.3 Phase shifting + spherical truncation

A related dealising is used by the DNS Earth Simulator code (Yokokawa et al., 2002). This requires computing the nonlinear term with only two phases (as opposed to the 8 phases required above), and then applying a spherical truncation:

$$\hat{\mathbf{u}}(l,m,n) \qquad \text{for } \sqrt{l^2+m^2+n^2} < \frac{\sqrt{2}}{3}N$$

With this type of dealiasing, we define the maximum wave number in our simulation to be $k_{\max} = 2\pi N\sqrt{2}/3$. Twice as many FFTs are needed, so the total cost is 18 3D FFTs (as

opposed to 9 3D FFTs for 2/3 rule). The total number of modes retained is $\frac{4}{3}\pi(\frac{\sqrt{2}}{3}N)^3$ Comparing this to the 2/3 rule above, one can see that for the same grid and twice the cost, this method does not retain twice as many modes. But if you compute cost per fully retained spherical shell $k_{\max}$, you can see this is the most efficient in that regards.

7.4 Spherical partial dealiasing

Spherical partial dealiasing does not exactly remove all aliasing errors from the nonlinear term. It instead relies on the fact that in fully developed turbulence, the energy will decay at a rate of $k^{-5/3}$. (where k is the spherical wave number, $k = 2\pi\|(l,m,n)\|$.) Under this assumption, bounds on the aliasing errors can be derived suggesting it is sufficient to dealias using:

$$\hat{\mathbf{u}}(l,m,n) = 0 \qquad \text{for } \sqrt{l^2+m^2+n^2} > \frac{\sqrt{2}}{3}N$$

With this type of dealiasing, we define the maximum wave number in our simulation to be $k_{\max} = 2\pi N\sqrt{2}/3$

7.5 Other dealiasing methods

There is also a phase shifting method which does not require a spherical truncation. This method is an exact dealiasing which does not result in any loss of resolution ($k_{\max} = \pi N$) but requires additional 3D FFTs. Originally developed by Patterson and Orszag (1971). It requires 8 times as many FFTs (in three-dimensions) as the 2/3 rule, but the FFTs are 2/3 the size. It is always more expensive than using the 2/3 rule. (Canuto et al., 1998, page 85, but this may not take into account the cost of message passing).

There is also the Rogallo (1977, 1981) scheme. Developed for a two stage time stepping scheme. The second stage uses a different (random) phase from the first, and the aliasing errors can then be shown to be reduced to $O(\Delta t^2)$. I dont know if any one has extended this to an RK4 scheme. This is the scheme used by the RK2 Georga Tech code.

Finally, Chuck Zemach (1994, personal notes) developed a phase shifting approach that exactly remove dealiasing for all wave numbers. Each inverse transform must be performed 4 times (with 4 different phases) and no changes are required for the forward transform. This gives a total of 27 3D FFTs (instead of 9 3D FFTs). It is cheaper than the Patterson and Orszag approach, but still quite expensive when compared with just two phases and spherical dealiasing.

8 Simulations of Turbulence

8.1 Relevant Parameters

We give some of the parameters used to discribe turbulent flow.

Velocity

$$\mathbf{u} = (u_1, u_2, u_3)$$

Kinetic energy

$$E = \frac{1}{2} < \mathbf{u} \cdot \mathbf{u} >= \frac{1}{2} \sum_i < u_i^2 >$$

For the isotropic case, we can take $E = \frac{3}{2} < u_1^2 >$.

Dissipation rate

$$\epsilon = \nu \sum_i < \nabla u_i \cdot \nabla u_i >$$

Kolmogorov length scale

$$\eta = \nu^{\frac{3}{4}} \epsilon^{-\frac{1}{4}}$$

Taylor length scale

$$\lambda^2 = \frac{< u_1^2 >}{< u_{1,1}^2 >} = 10 E \nu / \epsilon$$

Taylor Reynols number

$$R_\lambda = \frac{\lambda < u_1 >}{\nu} = E \sqrt{\frac{20}{3 \nu \epsilon}} = \sqrt{\frac{20}{3}} \frac{E}{(\epsilon \eta)^{2/3}} \tag{1}$$

Eddy turnover time

$$t_e = 2E/\epsilon \tag{2}$$

8.2 CFL based time step

If the time step used is based on the CFL condition, than at the end of each timestep one global collective is required (of a single real*8 value). In practice we perform two global collectives (a sum and a min) of 96 bytes of data. This is used for the CFL condition and other diagnostics.

The usual CFL condition is written as

$$\Delta t \leq C_0 \Delta x / \max(\mathbf{u})$$

For incompressible homogeneous isotropic turbulence, it is customary to write this in terms of the kinetic energy E, and maximum wave number $k_{\max}$,

$$\Delta t \leq C E^{-1/2} k_{\max}^{-1}$$

We have determined, via experiment, that a stable timestep is given by

$$C = 0.5.$$

As a check on this value, consider that given in Pope, *Turbulent Flows*, Cambridge University Press, 2000.

$$\Delta t = \frac{1}{20} \Delta x E^{-1/2} = .105 E^{-1/2} k_{\max}^{-1}$$

where we have assumed a 2/3 dealiasing ($k_{\max} = 0.66\pi/\Delta x$).

We write the time step in terms of eddy turnover time $t_e = 2E/\epsilon$ by

$$\Delta t / t_e = \frac{1}{2} C E^{-3/2} \epsilon k_{\max}^{-1}$$

8.3 Resolution Condition

Forced turbulence simulations, where one is primarily interested in the inertial range dynamics, are usually run with a resolution condition given by

$$k_{\max} \eta \geq 1$$

The Komogorov length scales gives an estimate of the length scales where the flow becomes dominated by viscosity effects and the dynamics beyond those lengths scales do not effect the larger, turbulent length scales.

More conservative investigators will sometimes require

$$k_{\max} \eta \geq 1.5$$

or even larger values for special applications.

This resolution condition determines the amount of viscosity ν needed. In practice, one chooses the smallest viscosity possible viscosity (for a given forcing) so that this resolution condition is not violated.

The dissipation rate ϵ is controlled by the forcing used. Once the forcing is determined, ϵ can be computated or estimated and then the appropriate diffusion coefficieint ν is determined from the resolution condition:

$$\nu = \epsilon^{1/3} \left(\frac{C_r}{k_{\max}} \right)^{4/3}$$

where $C_r = 1.0$ or the more conservative $C_r = 1.5$.

9 Forcing Examples

For the example calculations below, we will take the resolution condition $k_{\max}\eta = C_r$ and CFL constant $C = .5$. For convenience we rewrite Eq. **??** and Eq. **??** here in terms of our integral-like length scale $l = E^{3/2}/\epsilon$ and maximum wave number $k_{\max}$:

$$R_\lambda = 2.582\,(lk_{\max}/C_r)^{2/3} \qquad \Delta t/t_e = .5C(lk_{\max}/C_r)^{-1}$$

9.1 Deterministic Low Wave Number Forcing

The deterministic low wave number forcing we use is a version of Overholt and Pope, Comput. Fluids 27 1998. It uses a simplified version of their formula for the relaxation time scale, and only forces in the first two spherical wave numbers shells. It is documented in detail in Taylor, Kurien and Eyink, Phys. Rev. E 68, 2003.

This forcing results in a E and ϵ which are relatively insenstive to resolution. At $N = 1024$, we obtain $E = 1.9$ and $\epsilon = 3.6$, which gives an eddy turnover time of $t_e = 1.05$. At $N = 2048$, we get $E = 1.9$ and $\epsilon = 3.8$, which gives an eddy turnover time of 1.00. For our extrapolations to higher resolutions, we will use the later set of numbers.

Examples:

1. $N = 1024^3$, $L = 1$, Spherical dealiasing ($k_{\max} = 482(2\pi)$) $C_r = 1$ and $C = .5$:

$$\nu = 3.6 \times 10^{-5} \qquad R_\lambda = 420 \qquad t_e/\Delta t = 8300$$

2. $N = 2048^3$, $L = 1$, Spherical dealiasing ($k_{\max} = 965(2\pi)$), $C_r = 1$ and $C = .5$:

$$\nu = 1.4 \times 10^{-5} \qquad R_\lambda = 670 \qquad t_e/\Delta t = 17000$$

3. $N = 4096^3$, $L = 1$, Spherical dealiasing ($k_{\max} = 1930(2\pi)$), $C_r = 1$ and $C = .5$:

$$\nu = 5.6 \times 10^{-6} \qquad R_\lambda = 1063 \qquad t_e/\Delta t = 33000$$

4. $N = 12288^3$, $L = 1$, 2/3 rule dealiasing ($k_{\max} = 4096(2\pi) = 25736$), $C_r = 1$ and $C = .5$:

$$\nu = 2.1 \times 10^{-6} \qquad R_\lambda = 1800 \qquad t_e/\Delta t = 71000$$

5. $N = 12288^3$, $L = 1$, Spherical dealiasing ($k_{\max} = 5792(2\pi) = 36392$), $C_r = 1$ and $C = .5$:

$$\nu = 1.3 \times 10^{-6} \qquad R_\lambda = 2200 \qquad t_e/\Delta t = 100000$$

9.2 Deterministic Low Wave Number Forcing (Japanese Earth Simulator)

Similar to above. Deterministic, wave number shells 1 and 2. $E = 0.5, \epsilon = 0.0668$.

Examples:

1. $N = 4096^3$, $L = 2\pi$, Spherical dealiasing ($k_{\max} = 1930$), $C_r = 1$ and $C = .5$: (my formulas)
$$\nu = 1.7 \times 10^{-5} \qquad R_\lambda = 1200 \qquad t_e/\Delta t = 41000$$
Note: JES runs actually used $\Delta t = .25 \times 10^{-3}$, so 60,000 timesteps were required per eddy turnover time. The also give $\eta = .528 \times 10^{-3}$.

2. $N = 12288^3$, $L = 1$, 2/3 rule dealiasing ($k_{\max} = 4096$), $C_r = 1$ and $C = .5$:
$$\nu = 6.2 \times 10^{-6} \qquad R_\lambda = 2000 \qquad t_e/\Delta t = 87000$$

3. $N = 12288^3$, $L = 1$, Spherical dealiasing ($k_{\max} = 5792$), $C_r = 1$ and $C = .5$:
$$\nu = 3.9 \times 10^{-6} \qquad R_\lambda = 2500 \qquad t_e/\Delta t = 123000$$

9.3 Other Examples

1. Gotoh et al, 2002 with Stochastic forcing: $N = 1024^3$, $L = 2\pi$, Spherical dealiasing ($k_{\max} = 482$), $E = 1.79, \epsilon = .506$. Taking $C_r = 1$ and $C = .5$):
$$\nu = 2.1 \times 10^{-4} \qquad R_\lambda = 450 \qquad t_e/\Delta t = 9100$$
Published numbers: $R_\lambda = 460, nu = 2 \times 10^{-4}$.

2. Watanabe and Gotoh, 2004: $N = 1024^3$, $L = 2\pi$, Spherical dealiasing ($k_{\max} = 482$), $E = 1.97, \epsilon = .591$. Taking $C_r = 1$ and $C = .5$):
$$\nu = 2.2 \times 10^{-4} \qquad R_\lambda = 440 \qquad t_e/\Delta t = 9000$$
Published numbers: $R_\lambda = 427, nu = 2.4 \times 10^{-4}$.

3. Yeung 2005b: $N = 1024^3$, $L = 2\pi$, Spherical dealiasing ($k_{\max} = 482$), $E = 3.53, \epsilon = 1.302$. Taking $C_r = 1.5$ and $C = .5$):
$$\nu = 5.0 \times 10^{-4} \qquad R_\lambda = 360 \qquad t_e/\Delta t = 6500$$
Published numbers: $R_\lambda = 390, \nu = 4.37 \times 10^{-4}$.