TURBO manual

TURBO team

23 December 2008

CONTENTS

CHAPTER 1

INTRODUCTION TO TURBO

SECTION 1.1

EQUATIONS AND EQUATION PARAMETERS

The TURBO code is designed to solve the equations for an incompressible fluid in a three dimensional geometry with periodic boundary conditions in the three directions. More precisely, the balance equations that can be solved are the following :

$$\partial_t u_i = -\partial_j(u_i u_j) - \partial_i p + \nu\nabla^2 u_i + f_i + f_i^{\text{Lorentz}} \tag{1.1}$$

$$\partial_t b_i = -\partial_j(b_i u_j - u_i b_j) + \eta\nabla^2 b_i \tag{1.2}$$

$$\partial_t c_\alpha = -\partial_j(c_\alpha u_j) + \kappa_\alpha\nabla^2 c_\alpha + \sigma_\alpha(\{c_\beta\}) \tag{1.3}$$

where u_i is the velocity field, b_i is the magnetic field, c_α are passive scalar(s), each of which is characterized by a diffusion coefficient κ_α. Summation is assumed over repeated Latin style indices, but not over Greek style indices which correspond to passive scalar species. The kinematic viscosity is ν and the magnetic diffusivity is η. There is the possibility to include source or sink terms or even chemistry terms in the scalar equations through the function σ_α. Since periodic boundary conditions are assumed, the solutions to the balance equations are entirely determined by the initial conditions that are defined in the subroutine **`init_variables`** described in section 2.5.1. A fairly large variety of problems can be treated depending on the values given to some input parameters. These parameters are referred to as the equation parameters (NSCAL, EQTYPE, CHIM and FORCEPARA) and are defined in the namelist `equation_parameters` (see 2.2.1).

NSCAL: the number of passive scalars (α=1..NSCAL). It has to be a positive integer, otherwise the code stops with an error message.

`nscal` = 0 No passive scalar equation is solved.

`nscal` ≥ 1 `nscal` passive scalar equations have to be solved and `nscal` diffusion coefficients have to be provided as input parameters.

EQTYPE: the nature of the problem to be solved. It is assumed to be a positive integer, otherwise the code stops with an error message.

`eqtype` = 0 The velocity (1.1) and the magnetic field (1.2) equations are not solved. In that case, only the scalar equations (1.3), without the advection term $-\partial_j(c_\alpha u_j)$ will be considered (purely diffusive or diffusion-reaction problems).

`eqtype` = 1 The Navier-Stokes equation (1.1) will be solved without the Lorentz force f_i^{Lorentz} and the magnetic field equation is not solved.

`eqtype` = 2 The Navier-Stokes equation (1.1) will be solved with the Lorentz force f_i^{Lorentz} derived from the quasi-static approximation. The magnetic field equation is not solved.

`eqtype` = 3 The velocity (1.1) and the magnetic field (1.2) equations are both solved. The Lorentz force is given by $f_i^{\text{Lorentz}} = \partial_j b_i b_j$.

If NSCAL> 0 and EQTYPE> 0, the scalar equations (1.3) with the advection term will also be solved. CHIM: This parameter decides whether the chemical reaction terms have to be included. It is a boolean. If its value is "`TRUE`", the terms σ_α are computed through the subroutine `chemistry`, and if its value is "`FALSE`", the terms σ_α are ignored. The parameter CHIM is simply ignored if NSCAL=0. FORCEPARA : This parameter determines the nature of the forcing term f_i. It is assumed to be a positive integer, otherwise the code stops with an error message. The parameter FORCE is ignored if EQTYPE=0.

`forcepara` = 0 No forcing term is included. This is the typical choice for studying problems in which we are interested in the free evolution of a given initial condition.

`forcepara` = 1 The forcing corresponds to fixed energy and helicity injection rates in a shell of wave vectors.

`forcepara` = 2 The forcing imposes fixed energy and helicity levels in a shell of wave vectors.

`forcepara` = 3 The forcing corresponds to a constant (in time) force that is defined through the subroutine `external_forcing`.

`forcepara` = 4 The forcing corresponds to a constant (in time) force that corresponds to the Kologorov flow.

`forcepara` = 5 The forcing imposes a constant (in time) velocity profile defined through the subroutine `external_velocity`.

SECTION 1.2

GENERAL PROPERTIES OF DISCRETE FOURIER TRANSFORMS

1.2.1 THE GRID

The code TURBO computes the evolution of a number of fields that are driven by nonlinear partial differential equations. The number of fields depends on the problem defined by the parameter `eqtype`. In this section, such a typical field will be denoted $a(\vec{r})$. Because nonlinear terms are easy to compute when the field is known as a function of the position, while derivatives are easy to compute using the Fourier modes of the field, TURBO will use both the real space representation of this field $a(\vec{r})$ as well as its Fourier space representation $\tilde{a}(\vec{k})$. TURBO assumes that the field $a(\vec{r})$ is periodic in the three directions with periodicity length given by `lx`, `ly` and `lz`. The field is entirely defined by their behavior in a three dimensional periodic box $\texttt{lx} \times \texttt{ly} \times \texttt{lz}$ known as the computational domain. This computational domain is meshed using a grid of regularly spaced points with grid spacings that are entirely determined by the box sizes and the number of grid points in each direction which are given by `nx`, `ny` and `nz`:

$$\Delta_x = \frac{\texttt{lx}}{\texttt{nx}} \qquad \Delta_y = \frac{\texttt{ly}}{\texttt{ny}} \qquad \Delta_z = \frac{\texttt{lz}}{\texttt{nz}} \tag{1.4}$$

The domain size can also be used to define the smallest non trivial wave number in each direction:

$$k_x^0 = \frac{2\,\pi}{\texttt{lx}} \qquad k_y^0 = \frac{2\,\pi}{\texttt{ly}} \qquad k_z^0 = \frac{2\,\pi}{\texttt{lz}} \tag{1.5}$$

The code has these periodic boundary conditions, the grid structure and the wavevectors fully built in. The only parameters that the user can modify to change the grid are the box sizes `lx`, `ly`, `lz` and the number of grid points `nx`, `ny` and `nz`.

1.2.2 DISCRETE REAL SPACE REPRESENTATION

The discretized version of the field $a(\vec{r})$ is an array of real numbers that is denoted $a^d(\vec{i}) = a^d(i_x, i_y, i_z)$. General properties of the discrete representation of field using Fourier transforms are presented in this section and the actual storage of the arrays in TURBO is discussed in the following section. The vector of integers $\vec{i}$ corresponds to a position $\vec{r} = (x, y, z)$ on the grid according to:

$$(x, y, z) = (i_x\,\Delta_x, i_y\,\Delta_y, i_z\,\Delta_z) \tag{1.6}$$

Taking into account the periodicity of the field, positions corresponding to i_x and $i_x + \texttt{nx}$ are equivalent and the integers in $\vec{i}$ are assumed to be bounded as follows:

$$\begin{aligned} 0 \le i_x \le \texttt{nx} - 1 \\ 0 \le i_y \le \texttt{ny} - 1 \\ 0 \le i_z \le \texttt{nz} - 1 \end{aligned} \tag{1.7}$$

1.2.3 DISCRETE FOURIER SPACE REPRESENTATION

The discretized version of the field $\tilde{a}(\vec{k})$ is an array of complex numbers that is denoted $\tilde{a}^d(\vec{p}) = \tilde{a}^d(p_x, p_y, p_z)$. Again, the actual storage of this array in TURBO is discussed in the following section. The vector of integers $\vec{p}$ corresponds to a wave vector $\vec{k} = (k_x, k_y, k_z)$ defined as:

$$(k_x, k_y, k_z) = (p_x\,k_x^0, p_y\,k_y^0, p_z\,k_z^0) \tag{1.8}$$

Taking into account the grid structure, it is impossible to distinguish p_x and $p_x + \mathtt{nx}$. Indeed, a plane wave with a wave vector k_x takes exactly the same values on the grid as a plane wave with a wave vector $k_x + \mathtt{nx}k_x^0$, because, for each location x on the grid, one has

$$\exp\left(i\ k_x\ x\right) = \exp\left(i\ p_x\ i_x\ k_x^0\ \Delta_x\right) = \exp\left(i\ (p_x + \mathtt{nx})\ i_x\ k_x^0\ \Delta_x\right) \tag{1.9}$$

where the equalities $k_x^0\ \Delta_x\ =\ 2\ \pi/\mathtt{nx}$ and $\exp(\pm i\ 2\pi\ i_x) \equiv 1$ have been taken into account. The integers in $\vec{p}$ are thus assumed to be bounded as follows:

$$\begin{aligned} -\frac{\mathtt{nx}}{2} + 1 \leq p_x \leq \frac{\mathtt{nx}}{2} - 1 \\ -\frac{\mathtt{ny}}{2} + 1 \leq p_y \leq \frac{\mathtt{ny}}{2} - 1 \\ -\frac{\mathtt{nz}}{2} + 1 \leq p_z \leq \frac{\mathtt{nz}}{2} - 1 \end{aligned} \tag{1.10}$$

Actually, considering p_x any ranges of length $\mathtt{nx}$ would be acceptable but it is natural to keep only the smallest wave vectors. The same remark obviously holds for p_y and p_z. Finally, we note that since the field a is real, its Fourier representation has to satisfy the following constraint:

$$\tilde{a}^d(\vec{p})^* = \tilde{a}^d(-\vec{p})\,. \tag{1.11}$$

!!Warning!! : *These formulas implies that the number* $\mathtt{nx}$, $\mathtt{ny}$ *and* $\mathtt{nz}$ *must all be even.*

1.2.4 DISCRETE FOURIER TRANSFORMS

The numbers $a^d(\vec{i})$ and $\tilde{a}^d(\vec{p})$ are related through the expression for the discrete forward and inverse Fourier transforms:

$$\tilde{a}^d(\vec{p}) = C_k \sum_{\vec{i}} a^d(\vec{i})\ \exp[-si\ \vec{p}\cdot\vec{i}]\,, \tag{1.12}$$

$$a^d(\vec{i}) = C_r \sum_{\vec{p}} \tilde{a}^d(\vec{p}) \exp[+si\ \vec{p}\cdot\vec{i}]\,, \tag{1.13}$$

where C_r and C_k are normalization parameter and $s = \pm 1$ depending on the definitions adopted for the direct and inverse Fourier transform (the traditional definitions amounts to choosing $s = +1$). The quantity $\vec{p}\cdot\vec{i}$ must be here understood as:

$$\begin{aligned} \vec{p}\cdot\vec{i} &\equiv i_x\ p_x\ \Delta_x\ k_x^0 + i_y\ p_y\ \Delta_y\ k_y^0 + i_z\ p_z\ \Delta_x z\ k_z^0 \\ &= 2\ \pi\ (\frac{i_x\ p_x}{\mathtt{nx}} + \frac{i_y\ p_y}{\mathtt{ny}} + \frac{i_z z\ p_z}{\mathtt{nz}}) \end{aligned} \tag{1.14}$$

and the triple sums as:

$$\sum_{\vec{i}} \equiv \sum_{i_x=0}^{n_x-1} \sum_{i_y=0}^{n_y-1} \sum_{i_z=0}^{n_z-1}\,, \tag{1.15}$$

$$\sum_{\vec{p}} \equiv \sum_{p_x=-n_x/2+1}^{n_x/2} \sum_{p_y=-n_y/2+1}^{n_y/2} \sum_{p_z=-n_z/2+1}^{n_z/2}\,, \tag{1.16}$$

An important property of the inverse and direct Fourier transform is that, when applied successively on a field $a(\vec{x})$, they should leave it unchanged. This can be easily proven using the definitions (1.13) and (1.12):

$$\begin{aligned} a^d(\vec{i}) &= C_r \sum_{\vec{p}} \tilde{a}^d(\vec{p}) \exp\left(si\ \vec{p}\cdot\vec{i}\right) \\ &= C_r \sum_{\vec{p}} C_k \sum_{\vec{i}} a^d(\vec{i}') \exp\left(-si\ \vec{p}\cdot\vec{i}'\right) \exp\left(si\ \vec{p}\cdot\vec{i}\right) \\ &= C_r\ C_k \sum_{\vec{i}'} a^d(\vec{i}') \sum_{\vec{p}} \exp\left(si\ \vec{p}\cdot(\vec{i}-\vec{i}')\right) \end{aligned} \tag{1.17}$$

Now, we will use the following properties:

$$\sum_{\vec{i}} \exp\left(si\ \vec{p}\cdot\vec{i}\right) = N\delta_{\vec{p},0}\,, \tag{1.18}$$

$$\sum_{\vec{p}} \exp\left(si\ \vec{p}\cdot\vec{i}\right) = N\delta_{\vec{i},0}\,, \tag{1.19}$$

where $\delta_{\vec{p},0} = 1$ if $\vec{p} = (0,0,0)$ and 0 otherwise and $N = \texttt{nx ny nz}$. In particular, using the formula (1.18) in (1.17) yields:

$$a^d(\vec{i}) = N\ C_r\ C_k\ a^d(\vec{i})$$

which, of course, implies the important following constraint on the normalization factors:

$$N\ C_r\ C_k = 1\,. \tag{1.20}$$

SECTION 1.3

FIELD REPRESENTATIONS IN TURBO

1.3.1 REAL SPACE REPRESENTATION IN TURBO

The code is fully parallelized and can run on any number `numprocs` of processors. In practice, however, it is advised to use a number of processors that is simultaneously a power of 2 and a divisor of `nz`. Indeed, the computational grid is split into subdomains that correspond to slices in the z direction with `nx` $\times$ `ny` $\times$ `local_nz` grid points. When `numprocs` is a divisor of `nz`, the slices have all the same sizes and `local_nz=nz/numprocs` is the same on all the processors.

The discretized version of the field $a^d(\vec{i})$ is thus split into `numprocs` pieces and, on each processor, it is stored in a tri-dimensional real array which dimensions are defined by the following declaration statement:

```
REAL, DIMENSION(0:rx,0:ry,0:rz) ::  ar
```

The value of the parameters `rx`, `ry` and `rz` depend on the algorithm used for computing the discrete Fourier transforms. In the present distribution of TURBO the librayry `FFTW` is used and need to have `rx=nx+1`, `ry=ny-1` and `rz=local_nz-1`. The real number stored in `ar(ix,iy,iz)` is thus related to a position that depends on the integers `ix`, `iy` and `iz` as well as on the processor number `myid`:

$$\begin{aligned} x &= \mathtt{ix} * \mathtt{lx}/\mathtt{nx}, \\ y &= \mathtt{iy} * \mathtt{ly}/\mathtt{ny}, \\ z &= (\mathtt{local\_z\_start} + \mathtt{iz}) * \mathtt{lz}/\mathtt{nz}, \end{aligned} \tag{1.21}$$

The values of `local_nz` and `local_z_start` depend on the processor and are initialized in the subroutine **`init_fft3d()`** in the file *`tt_fft.f90`* and are output when the subroutine **`init_fft3d()`** is called.

The array `whereis_z(0:nz-1,2)` is defined to determine both the node and the value of `iz` that must be used to retrieve the grid point corresponding to $z = \mathtt{zz}\ \Delta_z$. The procedure is very simple and consists in going to the node `whereis_z(zz,1)` and get the value `(ix,iy,whereis_z(zz,2))`.

1.3.2 FOURIER SPACE REPRESENTATION IN TURBO

The discretized version of the Fourier modes $\tilde{a}^d(\vec{p})$ is also split into `numprocs` pieces. For efficiency reasons, the discrete Fourier transform algorithm that is used in TURBO (FFTW) produces, on each processor, an array of complex numbers that is defined by the following declaration statement:

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy) ::  ac
```

Here, `cx=nx/2`, `cz=nz-1` and `cy=local_ny_after_transpose-1`. Two important points have to be noted. First, the complete set of Fourier mode is now split in slices in the `ky` direction. Second, the order of the indices is (x, y, z) in real space while it is (k_x, k_z, k_y) in Fourier space. The complex number stored in `ac(qx,qz,qy)` is related to the Fourier mode that corresponds to a wave vector (`kx(qx)`, `kz(qz)`,`ky(qy)`). The first component of the wave vector is given by the simple rule:

$$\mathtt{kx}(\mathtt{qx}) = \mathtt{qx}\; k_x^0 \quad \text{for } \mathtt{qx} = 0 \text{ to } \mathtt{cx} = \mathtt{nx}/2 \tag{1.22}$$

Negative values of the wavevector `kx` are not needed because the symmetry of the Fourier representation of real fields ($u(-k) = u^*(k)$) is used in the code. Hence, only half of the Fourier modes are stored and the convention in TURBO is to keep only those correspnding to $\mathtt{kx} > 0$. The `kz` component is defined according to:

$$\mathtt{kz}(\mathtt{qz}) = \begin{cases} \mathtt{qz} & k_z^0 \quad \text{if } 0 \le \mathtt{qz} \le \mathtt{nz}/2 \\ (\mathtt{qz} - \mathtt{nz}) & k_z^0 \quad \text{if } \mathtt{nz}/2 < \mathtt{qz} \le \mathtt{cz} = \mathtt{nz} - 1 \end{cases} \tag{1.23}$$

It is interesting to notice that both `kx(qx)` and `kz(qz)` are defined independently of the processor. On the contrary, the `ky(qy)` component of the wave vector depends on both `qy` and the processor number `myid` according to the following rules:

$$\mathtt{ky}(\mathtt{qy}) = \begin{cases} \mathtt{qqy} & k_y^0 \quad \text{if } 0 \le \mathtt{qqy} \le \mathtt{ny}/2 \\ (\mathtt{qqy} - \mathtt{ny}) & k_y^0 \quad \text{if } \mathtt{ny}/2 < \mathtt{qqy} \le \mathtt{ny} - 1 \end{cases} \tag{1.24}$$

where `qqy=qy+local_y_start_after_transpose`. The values of the two parameters `local_ny_after_transpose` and `local_y_start_after_transpose` depend on the processor and are initialized in the subroutine **`init_fft3d()`** in the file *`tt_fft.f90`*. In order to ensure that the load of each processor is the same, it is advised to chose `numprocs` as a divisor of `ny`, in which case `local_ny_after_transpose` is independent of the processor and is simply given by `ny/numprocs`.

The array `whereis_ky(0:ny-1,2)` is defined to determine both the node and the value of `qy` that must be used to retrieve the the mode corresponding to $ky = \mathtt{yy}\; k_y^0$. The procedure is very simple and consists in going to the node `whereis_ky(yy,1)` and get the value `(qx,qz,whereis_ky(yy,2))`.

SECTION 1.4

THE DERIVATION OPERATOR

It is well known that the derivation operator is very easily implemented in terms of the Fourier coefficients. Considering the expression (1.13), it is clear that the Fourier coefficient of the gradient of the field $a(\vec{x})$ is given by $is\vec{k}\,\tilde{a}(\vec{k})$. There is however a slight difficulty with the discretized version of the derivative in the Fourier representation. Let us consider for instance a field that would have a single mode corresponding to the largest k_x mode only and let us denote this largest k_x by $k_x^{\max} = \pi\, n_x/\ell_x$. Taken into account the fact that $a(x)$ has to be real, this field would read:

$$a(x) = 2\,\Re\left(\tilde{a}(k_x^{\max})\right)\,\cos\left[\frac{\pi\, n_x\, x}{\ell_x}\right] - 2\,\Im\left(\tilde{a}(k_x^{\max})\right)\,\sin\left[\frac{\pi\, n_x\, x}{\ell_x}\right]$$

First, it can be noted that there is no way to determine the value of $\Im\left(\tilde{a}(k_x^{\max})\right)$ since, on the grid, it is multiplied by a function that is always 0. Indeed, the grid values of x are given by $i_x\,\ell_x/n_x$. The largest mode should thus be purely real. Moreover, the first order derivative of this purely real mode will itself be always 0 on the grid for the same reason. There is thus no way to distinguish between the derivative of a constant signal and the derivative of the mode $k_x^{\max}$. For this reason, the turbo code puts to 0 all the modes that correspond to $p_x = n_x/2$ or $p_y = n_y/2$ or $p_z = n_z/2$. This is done at the end the subroutine **`rk_endstep`** (2.8.9).

SECTION 1.5

PARSEVAL'S THEOREM

1.5.1 PARSEVAL'S THEOREM FOR DISCRETE REPRESENTATION OF THE FIELDS

We consider in this section how to compute the volume integral of a product of two fields using the discrete representation of the fields. The type of quantity we want to evaluate are given by:

$$Q = \frac{1}{V} \int d\vec{r}\, a(\vec{r})\, b(\vec{r})\,. \tag{1.25}$$

It is chosen to divide the integral by V because, in many cases, the user will be interested in computing the volume average of a product. In practice, the quantity Q is easily approximated in real space by:

$$Q = \frac{1}{N} \sum_{\vec{i}} a^d(\vec{i})\, b^d(\vec{i})\,, \tag{1.26}$$

where the volume $d\vec{r}$ has been approximated by V/N. However, in a spectral code, the fields are often expressed in the Fourier representation and transforming them into the real space representation is time consuming. It is thus interesting to express the quantity Q in terms of the Fourier representations:

$$\begin{aligned} Q &= \frac{C_r^2}{N} \sum_{\vec{i}} \sum_{\vec{p}} \sum_{\vec{p}} \tilde{a}^d(\vec{p})\, \tilde{b}^d(\vec{p})\, \exp\left[s\, i\, (\vec{p}+\vec{p}')\cdot\vec{i}\right] \\ &= \frac{C_r^2}{N} \sum_{\vec{p}} \sum_{\vec{p}'} \tilde{a}^d(\vec{p})\, \tilde{b}^d(\vec{p}')\, N\, \delta_{\vec{p},-\vec{p}'} \\ &= C_r^2 \sum_{\vec{p}} \tilde{a}^d(\vec{p})\, \tilde{b}^d(\vec{p})^*\,. \end{aligned} \tag{1.27}$$

where, we have used the property (1.19). Thi formula is the expression of the Parseval's theorem for the discrete representions of the fields. In most spectral code, only half of the Fourier modes are stored using the property (1.11). If, for instance, only the modes with `kx(qx)` $\geq$ `0` are stored, the quantity Q can be evaluated as

$$\begin{aligned} Q = C_r^2 &\sum_{p_x=0}^{n_x/2} \sum_{p_y=-n_y/2+1}^{n_y/2} \sum_{p_z=-n_z/2+1}^{n_z/2} \tilde{a}^d(\vec{p})\, \tilde{b}^d(\vec{p}))^* \\ + C_r^2 &\sum_{p_x=1}^{n_x/2-1} \sum_{p_y=-n_y/2+1}^{n_y/2} \sum_{p_z=-n_z/2+1}^{n_z/2} \tilde{a}^d(\vec{p})^*\, \tilde{b}^d(\vec{p})\,. \end{aligned} \tag{1.28}$$

which can be rewritten as follows

$$Q = C_r^2 \sum_{p_x=0}^{n_x/2} M(p_x) \sum_{p_y=-n_y/2+1}^{n_y/2} \sum_{p_z=-n_z/2+1}^{n_z/2} \Re\left(\tilde{a}^d(\vec{p})\, \tilde{b}^d(\vec{p}))^*\right) \tag{1.29}$$

where the real array $M(p_x)$ is defined as follows:

$$M(p_x) = \begin{cases} 1 & \text{if} \quad p_x = 0 \\ 2 & \text{if} \quad 1 \leq p_x \leq n_x/2 - 1 \\ 0 & \text{if} \quad \mathtt{px} = n_x/2 \end{cases} \tag{1.30}$$

The structure of $M(p_x)$ takes into account the fact that modes with $p_x = n_x/2$ are set to 0 as a consequence of the definition of the derivation operator.

1.5.2 PARSEVAL'S THEOREM IN TURBO

Two important properties of TURBO have to be taken into account in order to compute the expressions (1.26) and (1.29) with the code. First, only part of the discrete field are stored on each node. Second, only half of the modes in Fourier space (corresponding to $k_x \leq 0$) are stored by TURBO. We first consider the expression (1.26) which implies that two discrete real fields have been defined

```
REAL, DIMENSION(0:rx,0:rz,0:ry) ::  ar, br
```

Computing the partial sum on each node – it will be denoted `Qr_node` – is straightforward:

```
Qr_node=0
DO iz=0,rz
  DO iy=0,ry
    DO ix=0,rx
      Qr_node=Qr_node+ar(ix,iy,iz)*br(ix,iy,iz)
    END DO
  END DO
END DO
Qr_node=Qr_node/N
```

The order of the loops is chosen to optimize the computation in Fortran. The quantity stored in `Qr_node` corresponds to a partial sum over all the grid points associated to the node `myid`. The global sum is then easily computed using standard MPI instruction:

```
CALL mpi_allreduce(Qr_node,Q,1,MPI_MYREAL,MPI_SUM,MPI_COMM_WORLD,ierr)
```

Computing the expression (1.29) implies that two discrete complex fields have been defined

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy) ::  ac, bc
```

Computing the partial sum on each node – it will be denoted `Qc_node` – is also straightforward:

```
Qc_node=0
DO qy=0,cy
  DO qz=0,cy
    DO qx=0,cx
      Qc_node=Qc_node+REAL(ac(qx,qz,qy)*CONJG(bc(qx,qz,qy)))*MM(qx)
    END DO
  END DO
END DO
Qc_node=Qc_node*Cr^2
```

The order of the loops is again chosen to optimize the computation in Fortran. The quantity stored in `Qc_node` corresponds to a partial sum over all the Fourier modes associated to the node `myid`. The global sum is then easily computed using standard MPI instruction:

```
CALL mpi_allreduce(Qc_node,Q,1,MPI_MYCOMPLEX,MPI_SUM,MPI_COMM_WORLD,ierr)
```

The array `MM(0:cx)` is initialized in the subroutine **`initk`** according to the rules (1.30).

SECTION 1.6

KINETIC FORCING

1.6.1 SHELL FORCING WITH CONSTANT ENERGY AND HELICITY INJECTION RATES

If the forcing parameter `FORCEPARA=1`, a forcing term is added to the right-hand-side of the Navier-Stokes equation, which can be written in Fourier space as follows :

$$\vec{f}(\vec{k}) = \alpha(\vec{k})\ \vec{u}(\vec{k}) + \beta(\vec{k})\ \vec{\omega}(\vec{k})\,, \tag{1.31}$$

where the parameter $\alpha(\vec{k})$ and $\beta(\vec{k})$ are real. The vectors $\vec{u}(\vec{k})$ and $\vec{\omega}(\vec{k})$ represent the Fourier modes of the velocity and the vorticity respectively. An interesting property of this forcing is that it is local in Fourier space. It is thus quite easy to determine its effect in terms of energy or helicity. For each wave vector $\vec{k}$, the energy $e(\vec{k})$, the helicity $h(\vec{k})$ and the enstrophy $\Omega(\vec{k})$ are defined by:

$$e(\vec{k}) = \frac{1}{2}\ \vec{u}(\vec{k}) \cdot \vec{u}(\vec{k})^* \tag{1.32}$$

$$h(\vec{k}) = \Re\left(\vec{u}(\vec{k}) \cdot \vec{\omega}(\vec{k})^*\right) \tag{1.33}$$

$$\Omega(\vec{k}) = \frac{1}{2}\ \vec{\omega}(\vec{k}) \cdot \vec{\omega}(\vec{k})^* = k^2\ e(\vec{k}) \tag{1.34}$$

The last formula $\Omega(\vec{k}) = k^2\ e(\vec{k})$ is only valid for incompressible flows. Indeed, the velocity Fourier mode can be rewritten as follows $\vec{u}(\vec{k}) = \vec{u}^r(\vec{k}) + i\ \vec{u}^i(\vec{k})$ where $\vec{u}^r(\vec{k})$ and $\vec{u}^i(\vec{k})$ are real vectors that represent respectively the real and the imaginary part of $\vec{u}(\vec{k})$. The vorticity then reads

$$\vec{\omega}(\vec{k}) = i\vec{k} \times \vec{u}(\vec{k}) = i\ \vec{k} \times \vec{u}^r(\vec{k}) - \vec{k} \times \vec{u}^i(\vec{k})\,. \tag{1.35}$$

Both vectors $\vec{u}^r(\vec{k})$ and $\vec{u}^i(\vec{k})$, as a consequence of incompressibility, are orthogonal to $\vec{k}$. The norm of $\vec{\omega}(\vec{k})$ is then given by :

$$\Omega(\vec{k}) = \frac{1}{2}\ \vec{\omega}(\vec{k}) \cdot \vec{\omega}(\vec{k})^* = \frac{1}{2}\ |\vec{k} \times \vec{u}^i(\vec{k})|^2 + \frac{1}{2}\ |\vec{k} \times \vec{u}^r(\vec{k})|^2 \tag{1.36}$$

$$= \frac{1}{2}\ k^2\ |\vec{u}^i(\vec{k})|^2 + \frac{1}{2}\ k^2\ |\vec{u}^r(\vec{k})|^2 = k^2\ e(\vec{k}) \tag{1.37}$$

where $e(\vec{k})$ is given by $(|\vec{u}^r(\vec{k})|^2 + |\vec{u}^i(\vec{k})|^2)/2$. Without the incompressibility condition, it would no be possible to simplify $|\vec{k} \times \vec{u}^r(\vec{k})|^2$ into $k^2\ |\vec{u}^r(\vec{k})|^2$ since the $\sin$ of the angle between $\vec{k}$ and $\vec{u}^r(\vec{k})$ would have to be taken into account. The helicity can also be computed in terms of $\vec{u}^r$ and $\vec{u}^i$:

$$h(\vec{k}) = \Re\left(\left(\vec{u}^r(\vec{k}) + i\ \vec{u}^i(\vec{k})\right) \cdot \left(i\ \vec{k} \times \vec{u}^r(\vec{k}) - \vec{k} \times \vec{u}^i(\vec{k})\right)^*\right) \tag{1.38}$$

$$= \vec{u}^r(\vec{k}) \cdot \left(\vec{k} \times \vec{u}^i(\vec{k})\right) + \vec{u}^i(\vec{k}) \cdot \left(\vec{k} \times \vec{u}^r(\vec{k})\right)\,. \tag{1.39}$$

Remembering that $\vec{a} \cdot (\vec{b} \times \vec{c}) \leq |\vec{a}|\ |\vec{b}||\vec{c}|$ and $2\ |\vec{a}|\ |\vec{b}| \leq |\vec{a}|^2 + |\vec{b}|^2$, the important following inequality is easily established:

$$h(\vec{k}) \leq 2\ k\ e(\vec{k})\,. \tag{1.40}$$

The energy injection rate due to the forcing term is given by:

$$\epsilon_e^{\vec{k}} = \left.\frac{de(\vec{k})}{dt}\right|_f = \Re\left(\vec{u}(\vec{k}) \cdot \vec{f}(\vec{k})^*\right)$$

$$= 2\ \alpha_k\ e(\vec{k}) + \beta_k\ h(\vec{k}) \tag{1.41}$$

The helicity injection rate due to the forcing term is given by:

$$\epsilon_h^{\vec{k}} = \left.\frac{dh(\vec{k})}{dt}\right|_f = \Re\left(\vec{f}(\vec{k})\cdot\vec{\omega}(\vec{k})^* + \vec{u}(\vec{k})\cdot[-i\vec{k}\times\vec{f}(\vec{k})]^*\right)$$
$$= 2\Re\left(\vec{f}(\vec{k})\cdot\vec{\omega}(\vec{k})^*\right) = 2\,\alpha_k\,h(\vec{k}) + 4\,\beta_k\,k^2\,e(\vec{k}) \tag{1.42}$$

If the total energy injection rate ϵ_e and the total helicity injection rate ϵ_h are the control parameters, the forcing parameters α and β are defined as:

$$\alpha(\vec{k}) = \frac{1}{2\,N_f}\,\frac{4\,k^2\,e(\vec{k})\,\epsilon_e - h(\vec{k})\,\epsilon_h}{4\,k^2\,e(\vec{k})^2 - h(\vec{k})^2} \tag{1.43}$$

$$\beta(\vec{k}) = \frac{1}{N_f}\,\frac{e(\vec{k})\,\epsilon_h - h(\vec{k})\,\epsilon_e}{4\,k^2\,e(\vec{k})^2 - h(\vec{k})^2} \tag{1.44}$$

where N_f is the number of forced modes. This choice ensures that each of the N_f forced modes is submitted to a forcing mechanism that injects energy at the rate $\epsilon_e^{\vec{k}} = \epsilon_e/N_f$ and helicity at the rate $\epsilon_h^{\vec{k}} = \epsilon_h/N_f$. Now, we define $h(\vec{k}) = 2\,k\,e(\vec{k})\,s(\vec{k})$ and $\epsilon_h = 2\,k\,\epsilon_e\,\phi$. The coefficient $s(\vec{k})$ has to be in the range $[-1,+1]$. The limits ($s(\vec{k}) = \pm 1$) can only be reached when the velocity and the vorticity Fourier modes are perfectly aligned $\vec{\omega}(\vec{k}) = \pm k\vec{u}(\vec{k})$. With these changes of variables, the expression for the forcing parameters reduce to:

$$\alpha(\vec{k}) = \frac{\epsilon_e}{2\,N_f\,e(\vec{k})}\,\frac{1 - s(\vec{k})\,\phi}{1 - s(\vec{k})^2} \tag{1.45}$$

$$\beta(\vec{k}) = \frac{\epsilon_e}{2\,N_f\,k\,e(\vec{k})}\,\frac{\phi - s(\vec{k})}{1 - s(\vec{k})^2} \tag{1.46}$$

There is no obvious constraint on ϵ_e and ϕ. It is possible to imagine a forcing that would inject helicity and no energy during a certain period. However, such a forcing would lead asymptotically to a singularity. For instance, if the nonlinear interactions are neglected, the energy and helicity equations read:

$$\partial_t e(k) = \epsilon_e - 2\,\nu\,k^2\,e(k) \tag{1.47}$$

$$\partial_t h(k) = \epsilon_h - 2\,\nu\,k^2\,h(k) \tag{1.48}$$

The asymptotic solutions for these equations are given by $e_\infty(k) = \epsilon_e/(2\,\nu\,k^2$ and $h_\infty(k) = \epsilon_h/(2\,\nu\,k^2$. Since $h(\vec{k}) \leq 2\,k\,e(k)$, it seems reasonable to impose this constraint to the asymptotic solution of the linear problem and consequently we will assume $\epsilon_h \leq 2\,k\,\epsilon_e$, which implies $\phi \leq 1$. Interestingly, in the limit of "maximal" helicity injection rate $\phi = 1$, the forcing coefficients simplify into:

$$\alpha(\vec{k}) = \frac{\epsilon_e}{2\,N_f\,e(\vec{k})}\,\frac{1}{1 + s(\vec{k})} \tag{1.49}$$

$$\beta(\vec{k}) = \frac{\epsilon_e}{2\,N_f\,k\,e(\vec{k})}\,\frac{1}{1 + s(\vec{k})} \tag{1.50}$$

Since $s(\vec{k})$ should be close to 1 in that case, no divergence in the forcing parameter is expected. The case $\phi = -1$ would not cause any difficulty either.

1.6.2 SHELL FORCING WITH CONSTANT ENERGY AND HELICITY LEVELS

Another possibility is to impose the value of both the velocity and the helicity in a shell of wave vectors, which corresponds to the case `FORCEPARA=2`. This is easily achieved by transforming, after each time iteration, the velocity according to:

$$\vec{u}(\vec{k}) \rightarrow \alpha(\vec{k})\ \vec{u}(\vec{k}) + \beta(\vec{k})\ \vec{\omega}(\vec{k}) \tag{1.51}$$

As a consequence, the vorticity is transformed as:

$$\vec{\omega}(\vec{k}) \rightarrow \alpha(\vec{k})\ \vec{\omega}(\vec{k}) + \beta(\vec{k})\ k^2\ \vec{u}(\vec{k}) \tag{1.52}$$

Let us denote e^* and $h^* = 2\ k\ \phi\ e^*$ the levels for the energy and the helicity that have to be imposed. The parameters $\alpha(\vec{k})$ and $\beta(\vec{k})$ have thus to be chosen so that:

$$\begin{aligned} e^* &= \frac{1}{2}\ \Big(\alpha(\vec{k})\ \vec{u}(\vec{k}) + \beta(\vec{k})\ \vec{\omega}(\vec{k})\Big)^2 \\ &= (\alpha(\vec{k})^2 + \beta(\vec{k})^2\ k^2)\ e(\vec{k}) + \beta(\vec{k})\ \alpha(\vec{k})\ h(\vec{k}) \qquad (1.53) \\ h^* &= \Big(\alpha(\vec{k})\ \vec{u}(\vec{k}) + \beta(\vec{k})\ \vec{\omega}(\vec{k})\Big) \cdot \Big(\alpha(\vec{k})\ \vec{\omega}(\vec{k}) + \beta(\vec{k})\ k^2\ \vec{u}(\vec{k})\Big) \\ &= 4\ \alpha(\vec{k})\ \beta(\vec{k})\ k^2\ e(\vec{k}) + \Big(\alpha(\vec{k})^2 + \beta(\vec{k})^2\ k^2\Big)\ h(\vec{k})\,. \qquad (1.54) \end{aligned}$$

The solutions to this system require some simple algebraic manipulations that lead to:

$$\alpha(\vec{k})^2 = \frac{e^*\ \left([1 - \phi\ s(\vec{k})] + \sqrt{[1-\phi^2]\ [1 - s(\vec{k})^2]}\right)}{2\ e(\vec{k})\ [1 - s(\vec{k})^2]} \tag{1.55}$$

$$\beta(\vec{k}) = \frac{e^*\ [\phi - s(\vec{k})]}{2\ k\ e(\vec{k})\ [1 - s(\vec{k})^2]\ \alpha(\vec{k})} \tag{1.56}$$

An interesting limit is reached when maximal helicity is imposed in the forcing shell ($\phi = 1$):

$$\alpha(\vec{k})_{\phi=1} = \frac{\beta(\vec{k})_{\phi=1}}{k} = \sqrt{\frac{e^*}{2\ e(\vec{k})\ [1 + s(\vec{k})]}} \tag{1.57}$$

The energy injection rate imposed by this forcing is simply (independently of the value of $s(k)$ and ϕ:

$$\epsilon_e = \frac{e^* - e(\vec{k})}{dt} \tag{1.58}$$

and the helicity injection rate is

$$\epsilon_e = 2\ k\ \frac{\phi\ e^* - s(\vec{k})\ e(\vec{k})}{dt} = \frac{\phi\ h^* - h(\vec{k})}{dt} \tag{1.59}$$

SECTION 1.7

TIME EVOLUTION OF VARIABLES

The purpose of the time evolution algorithm is to estimate the value of $y(t_0 + h)$ considering that $y(t_0)$ is known. The parameter h is referred to as the time increment or time step. The subroutines described in section 2.8 are designed to achieve this objective and have two main purpose. The first one is to compute $y(t_0 + h)$ using a Runge-Kutta algorithm. This algorithm require the evaluation of the time derivatives of all the variables. Since these time derivatives contain nonlinear term which cannot be fully capture on the same grid as the linear term (aliasing), the second purpose of these subroutines is to eliminate as much as possible this aliasing error.

1.7.1 RUNGE-KUTTA SCHEMES

Time evolution in TURBO is based on a modified Williamson, third-order low storage Runge-Kutta method. This approach is explained here by considering that the balance equations solved by TURBO can be rewritten schematically as follows:

$$\dot{y} = -\nu\, y + N(y). \tag{1.60}$$

The linear term represents the dissipative transport and N contains all the remaining terms, including the nonlinearities and the possible mechanical forces (except de Coriolis force $-2\,\vec{\Omega} \times \vec{u}$ in systems subject to a solid body rotation; this force is treated separately as a linear terms in a slightly more complex scheme than the dissipative terms). By considering the change of variables $y(t) = \exp(-\nu(t - t_0))\; z(t)$, the equation 1.60 can be rewritten:

$$\dot{z} = e^{\nu(t-t_0)} N(z e^{-\nu(t-t_0)}) \equiv F(z, t). \tag{1.61}$$

Solving the equation for z is analytically equivalent to solving the equation for y. However, the numerical algorithms presented below and designed for the equation 1.61 have the advantage to produce the exact solution for the linear equation (N=0). This would not be true if they would be implemented directly to the equation 1.60. The algorithms we are considering are implemented using a multi-step method:

$$i = 1,\ 2, \ldots,\ n : \begin{cases} g_i &= g_{i-1} + \gamma_i\, F(z_{i-1}, t_0 + \chi_{i-1}\, h) \\ z_i &= z_{i-1} + \alpha_i\, h\, F(z_{i-1}, t_0 + \chi_{i-1}\, h) + \beta_i\, g_i\, h \end{cases} \tag{1.62}$$

with $z_0 = z(t_0) = y(t_0)$, $g_0 = 0$ and $\chi_0 = 0$. In the final step, z_n is supposed to be an accurate estimate for $z(t_0 + h)$. Of course, these algorithms can also be written in terms of the explicit form of $F(z, t)$:

$$g_i = g_{i-1} + \gamma_i\, e^{\nu\chi_{i-1}h}\, N(z_{i-1}\, e^{-\nu\chi_{i-1}h}) \tag{1.63}$$

$$z_i = z_{i-1} + \alpha_i\, h\, e^{\nu\chi_{i-1}h}\, N(z_{i-1}\, e^{-\nu\chi_{i-1}h}) + \beta_i\, g_i\, h \tag{1.64}$$

A very convenient change of variables can be used to simplify these formula: $y_i = z_i\, e^{-\nu\chi_i h}$ and $k_i = g_i\, e^{-\nu\chi_i h}$. This leads to:

$$k_i = e^{-\nu\xi_i h}\ (k_{i-1} + \gamma_i\, N(y_{i-1})) \tag{1.65}$$

$$y_i = e^{-\nu\xi_i h}\ (y_{i-1} + \alpha_i\, h\, N(y_{i-1})) + \beta_i\, k_i\, h \tag{1.66}$$

where $\xi_i = \chi_i - \chi_{i-1}$. In these schemes, k_0 and χ_0 are both assumed to vanish and $\chi_n = 1$. The other parameters have to be chosen so that the difference between the actual solution $y(t + h)$ and y_n is of

order h^{p+1}. The scheme is then of order p. TURBO proposes two third-order accurate Runge-Kutta schemes.

The first one is based on three sub-steps and used the so-called "two-third" dealiasing method. The choice for the other parameter is not unique. We propose to consider only schemes for which the three sub-steps are evaluated at t_0, $t_0 + h/3$ and $t_0 + 2/3$. There is no particular justification for this choice except that it simplifies the procedure since all the ξ_i are equals to $1/3$. This choice is still compatible with a three-parameter family of coefficients that are defined by the following equalities.

$$\begin{array}{lll} \alpha_1 = -\beta_1\,\gamma_1 + 1/3 & \beta_2 = -1/(3\,\gamma_1) & \gamma_2 = -8\,\gamma_1/3 \\ \alpha_2 = -2/9 & \beta_3 = +1/(4\,\gamma_1) & \gamma_3 = -(4\,\alpha_3 - 3)\,\gamma_1 \end{array} \tag{1.67}$$

TURBO is implemented with the following particular choice and these parameters are defined in the subroutine **`rk3_twothird`** 2.8.2:

$$\begin{array}{lll} \alpha_1 = +1/2 & \beta_1 = +1/3 & \gamma_1 = -1/2 \\ \alpha_2 = -2/9 & \beta_2 = +2/3 & \gamma_2 = +4/3 \\ \alpha_3 = +1 & \beta_3 = -1/2 & \gamma_3 = +1/2 \end{array} \tag{1.68}$$

The second Runge-Kutta method is a four substep algorithm. The idea behind this approach is that all the computation of the nonlinear term have to appear in the final form of y_n with the same weight (which is then 1/4, as imposed by the condition that the first order of the expansion of y_n matches the first order of the analytical solution). The purpose of this additional constraint is to have for each couple of time iteration, heigh successive evaluation of the nonlinear terms that will be computed on shifted grids for alias removal 1.7.2. The grids have to be shifted by $\pm\Delta/2$ in each direction and we thus have height possible combinations in a three dimensional computation. These constraints imposes that $\chi_1 = \chi_2 = \chi_3 = 2/3$ but are compatible with a four parameter family of coefficients. We do not present the general form of this family of coefficients and show only the value actually implemented in TURBO:

$$\begin{array}{llll} \alpha_1 = +1/3 & \beta_1 = +1/3 & \gamma_1 = 1 & \chi_1 = 2/3 \\ \alpha_2 = -3/8 & \beta_2 = -3/4 & \gamma_2 = -3/2 & \chi_2 = 2/3 \\ \alpha_3 = +1/4 & \beta_3 = +1/2 & \gamma_3 = 0 & \chi_3 = 2/3 \\ \alpha_4 = +1/4 & \beta_4 = -1/6 & \gamma_4 = 0 & \chi_4 = 1 \end{array} \tag{1.69}$$

These parameters are defined in the subroutine **`rk4_shift`** 2.8.3:

1.7.2 DEALIASING METHODS

The aliasing error takes its origin in the property 1.9 that a plane wave with a wave vector k_x takes exactly the same values on the grid as a plane wave with a wave vector $k_x + \mathtt{nx}\ k_x^0$. When the nonlinearities are computed, this aliasing error becomes a serious issue. Let us considered a one dimensional problem with a signal s represented by modes with $k = k_0\ (-n/2+1), \cdots, k_0\ n/2$. The square s^2 has modes that correspond to $k = k_0\ (-n+2), \cdots, k_0\ n$. As a consequence, the mode of s^2 that corresponds to $k = k_0\ (-n+2)$ is undistinguishable from the mode $k = 2$. Two methods are considered to eliminate this difficulty.

The first one consists in i) assuming that only the modes of s with $k = k_0\ (-m/2+1), \cdots, k_0\ m/2$ are non zero and ii) imposing that these modes remain zero during the simulation. Let us consider a mode of s that corresponds to $k = q\ k_0$ with $0 \leq q < m/2$. If $2q > n/2$, this mode corresponds to a mode $2k$ in s^2 that will create aliasing in the mode $2q - n$ that is between $-n/2$ and $m - n$. However, if $m - n < -m/2 + 1$ the mode that is contaminated belongs to the range of modes that are imposed to remain zero and the aliasing error is eliminated. The largest value of m compatible with this inequality $3m/2 < n + 1$ is $m = 2/3n$. The two-third method thus consist simply in keeping all the modes outside the range $k = k_0\ (-3n/4 + 1), \cdots, k_0\ 3n/4$ to zero.

The second method is based on the following property. If the grid is shifted by a distance d, all the positions are simply shifted $x \rightarrow x + d$ and the modes are then multiplied by $\exp(i\ d\ q\ k_0)$. The aliasing error can be removed by i) shifting the grid by d, ii) compute the nonlinearity on this grid, iii) shift the grid by $-d$. Let us consider a mode $k = q\ k_0$ with $2q > n/2$. The contribution from aliasing is multiplied by $\exp(i\ d\ q\ k_0)$ in step i). Hence, the contribution of this term to the quadratic nonlinearity (computed in step ii), is multiplied by $\exp(2\ i\ d\ q\ k_0)$. Finally, when shifted back to the original grid, this term multiplied by $\exp(-i\ d\ (2q - n)\ k_0)$ in step iii) since this nonlinear product is stored on the mode $(2q - n)$. The total factor that multiplies the aliasing error is thus $\exp(i\ d\ n\ k_0)$. It is important to notice that the contributions that does not lead to aliasing errors are unaffected by this procedure.

If d is $\pi/(nk_0) \equiv \Delta$, the aliasing error is multiplied by $\exp(i\pi) = -1$. The idea is thus to perform two separate computations of the nonlinear term. These computations use respectively shifts $d = +\Delta/2$ and $d = -\Delta/2$ so that the aliasing error is opposite to each other in the two computations. Summing the two computations thus leads to an alias free nonlinear term. In three dimensional systems, height evaluations with different shifts are needed to get an alias-free computation of the nonlinear terms. This is of course quite prohibitive. The policy adopted in TURBO is to compute these height nonlinear as substep of a Runge-Kutta scheme. More precisely, two time steps of a four-substep Runge Kutta scheme are used in which the updated variable is, at lowest order, given by:

$$y_{n+1} = y + (F_1 + F_2 + F_3 + F_4)/4 * dt + \cdots \tag{1.70}$$

where the F_i's are the nonlinear terms computed at each substep on a different grid. If the same dt is used for two time steps, this scheme allows to perform the height evaluations with two time iterations. Of course, the F_i's are not evaluated exactly at the same time, so the aliasing error is only removed up to order dt. However, a random shift is added to the procedure so that the remaining aliasing error is multiplied by a random number, which should reduce further its impact on the accuracy.

SECTION 1.8

SOLID BODY ROTATION

Rotation effects due to a global solid body rotation of the reference frame have been implemented in TURBO. The solid body rotation is fully characterized by the vector $\vec{\Omega}$. Its direction defines the axis of rotation and its intensity gives the angular speed. Two accelerations appear in the Navier-Stokes equation due to the rotation:

$$\partial_t \vec{u} + \vec{u} \cdot \nabla \vec{u} \Rightarrow \partial_t \vec{u} + \vec{u} \cdot \nabla \vec{u} + 2\,\vec{\Omega} \times \vec{u} + \vec{\Omega} \times (\vec{\Omega} \times \vec{r}) \tag{1.71}$$

Hence, a new term, referred to the Coriolis force $-2\,\vec{\Omega} \times \vec{u}$, has to be added in the right hand side of the Navier-Stokes equations. The last term is known as the centripetal acceleration. It seems to break the translation invariance of the Navier-Stokes equations since it depends on the position $\vec{r}$. However, in incompressible fluids, the centripetal acceleration term can be combined with the pressure by virtue of the equality $\vec{\Omega} \times (\vec{\Omega} \times \vec{r}) = -\nabla(\vec{\omega} \times \vec{r})^2/2$. Hence, the only visible effect of the rotation is the Coriolis force. Although there is no restriction in the implementation of rotation in TURBO regarding the direction of $\vec{\Omega}$, we will assume here that $\vec{\Omega}$ is along the z axis in order to simplify the present discussion. In terms of components, this additional force can be expressed as $-2\,\Omega\,\epsilon_{i3l}\,v_l$. This expression is not divergence free. However, in Fourier space, it is fairly simple to keep only the divergence free part of it by using the projector $P_{ij} = \delta_{ij} - k_i k_j/k^2$. The Navier-Stokes equations can then be written in Fourier space as follows

$$\partial_t u_i = -\nu k^2 u_i - 2\,\Omega\,P_{ij}\epsilon_{j3l}u_l + n_i(u_m) \tag{1.72}$$

Here, n_i is the divergence free non-linear terms. This equation can be re-written as follows

$$\partial_t u_i = L_{ij}u_j + n_i(u_m) \tag{1.73}$$

where $L_{ij} = -\nu k^2 \delta_{ij} - 2\,\Omega\,B_{ij}$ and $B_{ij} = P_{il}\epsilon_{l3j}$. The solution of the linear equation is given by:

$$u_i(t_0 + \tau) = C_{ij}(\tau)\,u_j(t_0) \tag{1.74}$$

where

$$C_{ij}(\tau) = e^{\tau\,L_{ij}} = e^{\tau(-\nu k^2\delta_{ij} - 2\Omega\,B_{ij})} = e^{-\tau\nu k^2} D_{ij} \tag{1.75}$$

and $D_{ij} = e^{-2\tau\Omega\,B_{ij}}$. Here, we have used the fact that the exponential of a sum of two matrices $M_1 + M_2$ is the product of the exponentials $e^{M_1}\,e^{M_2}$ when M_1 and M_2 commute. This is obviously the case for B_{ij} and δ_{ij} since δ_{ij} commutes with any matrix. Introducing $\omega = 2\,\Omega\,k_3/k$, the following explicit form for the matrix D_{ij} can be derived: $D_{ij} = \tilde{D}_{ij} + a_i k_j$ where

$$\tilde{D}_{ij}(\omega\tau) = \cos\omega\tau\,\delta_{ij} - \sin\omega\tau\,\frac{k_l}{k}\,\epsilon_{ijl} = \frac{1}{k}\begin{pmatrix} k\,\cos\omega\tau & -k_3\,\sin\omega\tau & k_2\,\sin\omega\tau \\ k_3\,\sin\omega\tau & k\,\cos\omega\tau & -k_1\,\sin\omega\tau \\ -k_2\,\sin\omega\tau & k_1\,\sin\omega\tau & k\,\cos\omega\tau \end{pmatrix} \tag{1.76}$$

$$a_i = \frac{1}{k\,k_3}\begin{pmatrix} -k_2\,\sin\omega\tau \\ k_1\,\sin\omega\tau \\ k\,(1-\cos\omega\tau) \end{pmatrix} \tag{1.77}$$

Since the velocity is divergence free, the term $a_i k_j$ can be neglected in 1.74. The following change of variables can then be introduced:

$$u_i = e^{-\tau\nu k^2}\,\tilde{D}_{ij}(\omega\tau)\,z_j \tag{1.78}$$

The new vector z_j is also assumed to be divergence free ($k_i z_i = 0$). It is quite simple to show that the complete inverse of the matrix $\tilde{D}_{ij}(\omega\tau)$ can be expressed as follows:

$$\tilde{D}^{-1}_{ij}(\omega\tau) = \tilde{D}_{ij}(-\omega\tau) + \frac{\sin^2\omega\tau}{\cos\omega\tau}\,\frac{k_i k_j}{k^2} \tag{1.79}$$

Hence, when considering only their action on divergence free vectors, the matrices $\tilde{D}_{ij}(\omega\tau)$ and $\tilde{D}_{ij}(-\omega\tau)$ can be considered as the inverse of each other since the last term in 1.79 vanishes. The evolution equation for u_i can now be transformed into:

$$\partial_\tau u_i = (-\nu k^2\delta_{ij} - 2\,\Omega\,B_{ij})u_j + n_i(u_m) \tag{1.80}$$

$$= e^{-\tau\nu k^2}\,\tilde{D}_{ij}(\omega\tau)\partial_\tau z_j - \nu k^2 u_i + e^{-\tau\nu k^2}\left(\partial_\tau \tilde{D}_{ij}(\omega\tau)\right) z_j \tag{1.81}$$

which reduces to:

$$\partial_\tau z_i = e^{+\tau\nu k^2}\;\tilde{D}^{-1}_{ij}(\omega\tau)\;n_j(u_m)$$

where we have used the following relation that can be easily verified:

$$2\,\Omega\,B_{il}\tilde{D}_{lj}(\omega\tau) = -\partial_\tau\tilde{D}_{ij}(\omega\tau) + b_i k_j$$

where $\vec{b} = -\omega\,\sin\omega\tau\,\vec{k}/k^2 + 2\,\cos\omega\tau\,\vec{\Omega}\times\vec{k}/k^2$. Assuming that the nonlinear term is divergence free, or that its divergence has been taken out, the evolution equation for the variables z_i can now be written as follows:

$$\partial_\tau z_i = e^{+\tau\nu k^2}\;\tilde{D}_{ij}(-\omega\tau)\;n_j(u_m) \tag{1.82}$$

and the changes of variables are given by:

$$z_i = e^{+\tau\nu k^2}\;\tilde{D}_{ij}(-\omega\tau)\;u_j \tag{1.83}$$

$$u_i = e^{-\tau\nu k^2}\;\tilde{D}_{ij}(+\omega\tau)\;z_j \tag{1.84}$$

It can be noted, that the expression (1.76) does not depends on the assumption that the rotation vector is aligned along the z axis. Hence, it the expression for $\tilde{D}(\omega\tau)$ is valid in general if the frequency ω is re-defined as follows: $\omega = 2\,\vec{\Omega}\cdot\vec{k}\;/\;k$.

In practice, in the presence of a solid body rotation, the velocity and its associated work array must be multiplied by the matrix $\tilde{D}(\omega\tau)$ at each substep in the Runge-Kutta scheme 1.65-1.66. It must be noted that the magnetic field and the scalar fields evolution equations are not modified by the rotation [see for instance : Numerical simulations of rotating sunspots, by G.J.J. Botha, A.M. Rucklidge, F.H. Busse and N.E. Hurlburt. In 10 Years of SOHO and Beyond (eds. D. Spadaro, B. Fleck and J.B. Gurman) ESA SP-617: Noordwijk (2006)].

CHAPTER 2

CORE TURBO SOURCE FILES

This chapter proposes a complete description of the programs, subroutines and modules that are part of the core solver in the TURBOdistribution package. The core solver requires only the files with a name starting by `tt_` that should not be modified by the user and the files with a name starting by `xx_` which is the place where user-defined subroutines should be introduced. The present release of the standard distribution package of TURBO also comtains a number of files with the prefix `pp_` which correspond to post-processing programs. The documentation for this file will be added in a future release.

For each file, all the subroutines are described in the same order as they appear in the code. This documentation is designed to users who have some knowledge of the problems described in the preceeding chapters and have a basic knowledge of Fortran and a minimal understanding of MPI.

SECTION 2.1

THE PROGRAM IN FILE: TT_MAIN.F90

This is the main program of the TURBO solver. The structure of the program can be summarized in the following scheme:

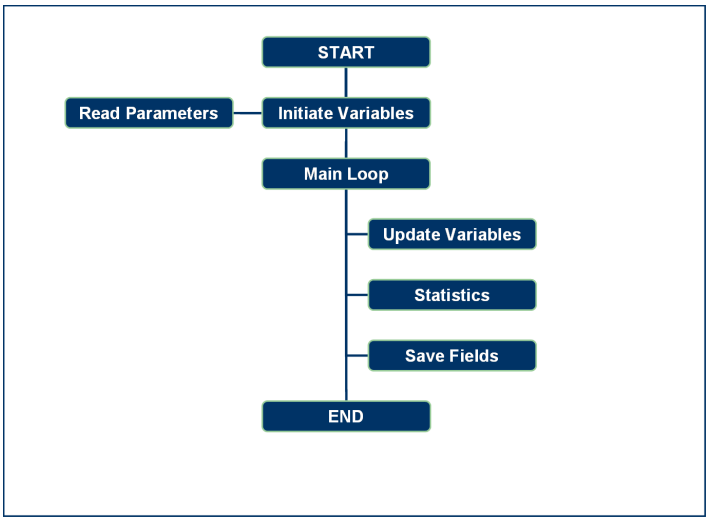

Figure 2.1: Typical TURBO run. Variables are initialized depending on the type of simulation defined in the parameter file. In the main loop, the variables are updated, the statistics are computed and the fields are saved at the desired intervals. Some final actions are then taken before exiting.

The main program has three parts that correspond respectively to i a number of initializations, ii the main loop and iii several actions that have to be taken before exiting.

The first part starts by **initializing** the MPI parameters and variables. It then reads the name of the simulation `simuname` and the corresponding parameter file `simuname.par` and time & save parameter `simuname.n` file. The initial iteration number is set to `niter_done+1` and the final iteration number is set to the minimum between the total number of iterations requested for the simulation `niter_todo` and the sum `niter_done+nstep`. It is important here to make the difference between the "simulation" and the "run". The complete numerical study of a problem will be referred to as the simulation. For practical reasons (computer queueing system limitations, for instance), it might be needed to split the simulation into several runs. The number of iterations needed for the simulation is `niter_todo` while the number of iterations in a run is `nstep`. In this first part, the main program also initializes the variables and the statistics through the call to the subroutine **`init_variables`**. Finally, it outputs the files (**`output`**) but not the statistics and starts computing the statistics: **`external_stats_compute`**.

The **main loop** is very simple. It computes the time step every two iterations. The time step is automatically adapted every two time step only as imposed by the grid-shifting dealiasing procedure implemented in one of the Runge-Kutta schemes. The variables are then updated every time step. The required statistics are computed every `nstat` iterations and the files are saved every `nsave` iterations. The parameters `nstat` and `nsave` are part of the namelist `save_parameters` in the parameter file.

Before **exiting**, the main program computes the statistics and outputs the variables one last time if needed. It then tests if the end of the run corresponds to the end of the simulation. If not, it creates an empty file *`simuname.go`*. The existence of this empty file can then be used to automatically relaunch a run in the queueing system using an appropriate script file. The program then deallocates the memory and exits.

SECTION 2.2

THE MODULES IN FILE: TT_MOD.F90

This file contains the different modules used by TURBO. The modules are used to define global variables and parameters used by the code. These global variables and parameters are accessible by any subroutine which includes at the beginning the statement `USE "module_name"` where `module_name` refers to the name of the module in which the variable is declared. When a module is used by a subroutine, it is not allowed to define locally another variable with the same name as a variable defined in the module. The modules used in TURBO are:

- `parameters`: contains the simulation parameters.
- `variables`: contains the simulation field variables.
- `fftwmodule`: is needed for the Fast Fourier Transform subroutines.
- `mpi_var`: is needed for the MPI processes.
- `statmod`: used for external defined subroutines.

2.2.1 MODULE PARAMETERS

This module contains a large number of variables (quantities that can be changed in the subroutines) and parameters (quantities that are defined and have a fixed value assigned in the module) used throughout many subroutines in TURBO.

USEFUL CONSTANT PARAMETERS

The module `parameters` first defines simple parameters such as :

- `pi` = 3.1415...
- `EPS` = 10^{-10} (used as a very small number...)
- `twopi` = 2π
- `I` = $\sqrt{-1}$
- `namelength` = 6 (defines the number of characters used for the simulation name)
- `SIZEOF_REAL = KIND(1.0)` (used to determine if the code has been compiled in single of double precision).

USEFUL VARIABLES AND ARRAYS

Next, the module `parameters` is used to declare variables and arrays that are used in more than one subroutine. The first three real numbers `rsx`, `rsy` and `rsz` are random shifts that are used for two successive time steps if random shifts to remove the aliasing error 1.7. The integers `plan_CtoR` and `plan_RtoC` are needed by the FFTW subroutines. The character string `simname` designates the simulation name and is mainly used in the input and output subroutines.

Also, various arrays defined in the subroutine **`init_wavevectors`** (2.5.3) are declared here : `MM`, `kx`, `ky`, `kz`, `ikx`, `iky`, `ikz`, `ksquare`, `knorm`, `km2` and `zeros_twothird`.

The module `parameters` also contains a number of namelists. A namelist is a facility for grouping variables for input and output. All the data that have to be specified by the user in order to define the type of runs are grouped using namelists in the file *`simname.par`* where "simname" is the name of the simulation. These namelists are defined in the module `parameters`:

NAMELIST "DIM_AND_SIZES" AND RELATED QUANTITIES

This namelist contains three integers `nx, ny, nz` representing the numbers of grid points in each direction and three real numbers `lx, ly, lz` corresponding to the sizes of the computational domain. Other variables closely related to these quantities and initialised in the subroutine **`init_fft3d`** are also defined after the definition of the namelist `dim_and_sizes`:

- `Ns=nx*ny*nz`, the total number of grid points
- `Cr=1.0/Ns` and `Ck=1.0` normalisation factors for the FFT
- `cx,cy,cz` the dimension of complex arrays (may depend on the computational node).
- `rx,ry,rz` the dimension of real arrays (may depend on the computational node).
- `dx,dy,dz` the grid sizes (`dx=lx/nx`, ...), initialised in the subroutine **`init_wavevectors`**
- `kMx,kMy,kMz` the largest wave vector in each direction (`kMx=`π`nx/lx`, ...), initialised in the subroutine **`init_wavevectors`**. The smallest of these three values is retained to define `kM`.
- `k0x,k0y,k0z` the smallest wave vector in each direction (`k0x=`2π`/lx`, ...), initialised in the subroutine **`init_wavevectors`**. The largest of these three values is retained to define `k0`.

NAMELIST EQUATION_PARAMETERS AND RELATED QUANTITIES

This namelist contains four integers. The first integer is FORCEPARA. It defines the type of forcing – if any – that will be used in the subroutine **forcing** 2.4.1. It can take any of the following values:

- 0 : No forcing term is added in the Navier-Stokes equation.
- 1 : The forcing injects fixed energy and helicity rates in a shell of wave vectors 1.6.1.
- 2 : The forcing imposes fixed energy and helicity levels in a shell of wave vectors 1.6.2.
- 3 : The forcing is defined by the user in an external subroutine external_forcing.
- 4 : Kolmogorov forcing.
- 5 : Imposes a number of constraints on the velocity field defined by the user in the external subroutine **external_modif_vel**.

The second integer is CHIM. It defines whether reactive terms are added (CHIM=1) in the passive scalar equations or not (CHIM=0).

The third integer is NSCAL. It defines the numbers of passive scalar equations. It can take integer values between 0 and 99.

The fourth integer is EQTYPE. It defines the type of problem that has to be solved by TURBO. It can take one of the following values:

```
INTEGER, PARAMETER ::  reac_diff     = 0
INTEGER, PARAMETER ::  navier_stokes = 1
INTEGER, PARAMETER ::  qs_mhd        = 2
INTEGER, PARAMETER ::  full_mhd      = 3
```

The parameters reac_diff, navier_stokes, qs_mhd and full_mhd are introduced to help in the reading of the code. For instance, if EQTYPE=1, the convention in TURBO assumes that the code is solving the Navier-Stokes equations (without MHD terms). It is easier however to understand the exact meaning of a test in the code if this test is written IF (EQTYPE.eq.navier_stokes) THEN ... rather than IF (EQTYPE.eq.1) THEN ... Also, three logicals are defined:

- VELEQ : True if the velocity equation has to be solved.
- MAGEQ : True if the magnetic field equation has to be solved.
- SCAEQ : True if scalar field equation(s) has(ve) to be solved.

NAMELIST ROT_PARAMETERS AND RELATED QUANTITIES

This namelist contains three real numbers Omega_x, Omega_y, Omega_z that correspond to the three components of a possible solid body rotation vector. Immediately after the declaration of this namelist, the module parameters also declares the logical ROTEQ. In the subroutine **read_para**, the value of these numbers are read and checked. If all three real numbers are zero (in practice, if their absolute value is lower than EPS), TURBO considers that no rotation effect has to be included and sets the logical ROTEQ to false otherwise it is set to true.

NAMELIST VEL_PARAMETERS AND RELATED VARIABLES

This namelist is read in the subroutine **read_para** only if the logical VELEQ is true. Otherwise, it is simply ignored. It contains one integer and several real numbers. The integer init_vel determines

the type of initialisation procedure used for the velocity at the beginning of the run. It can take the following values:

```
INTEGER, PARAMETER ::  restart      = 0
INTEGER, PARAMETER ::  random       = 1
INTEGER, PARAMETER ::  taylor_green= 2
INTEGER, PARAMETER ::  extern       = 3
```

Again, the parameters `restart`, `random`, `taylor_green` and `extern` are introduced to help in the reading of the code.

The real number `nu` is the molecular viscosity. Because the mode corresponding to $k_x = k_y = k_z = 0$ are kept constant by the Navier-Stokes equation in absence of average body force, they have to be prescribed by the users using the real numbers `u0_x`, `u0_y` and `u0_z`.

The last four real numbers `vel_a`, `vel_b`, `vel_c` and `vel_d` are used to define the spectrum used if the velocity field is initialised using **`init_random_vec_divfree`**. The first two of these numbers are also used to define the amplitude and the orientation of the Taylor Green vortex when `init_vel=taylor_green`.

NAMELIST MAG_PARAMETERS AND RELATED VARIABLES

This namelist is read in the subroutine **`read_para`** only if the logical `MAGEQ` is `true`. Otherwise, it is simply ignored. It contains one integer `init_mag` which is used exactly as `init_vel`. It also contains the magnetic diffusivity `eta`, the three modes `b0_x`, `b0_y` and `b0_z` corresponding to $k_x = k_y = k_z = 0$ and four real numbers `mag_a`, `mag_b`, `mag_c` and `mag_d` that define the spectrum used if the magnetic field is initialised using **`init_random_vec_divfree`**.

NAMELIST SCA_PARAMETERS AND RELATED VARIABLES

For each passive scalar, TURBO requires one integer and 5 real numbers. The integer plays the role of `init_vel` and determines which type of initialisation is used. The first real number is the scalar diffusivity and the last four real numbers are used to initialise the scalar field and play the same role as `vel_a`, `vel_b`, `vel_c` and `vel_d`. However, namelists can not contain allocatable arrays. For that reason, the namelist `sca_parameters` contains 6 arrays of dimensions `(1:99)`: `init_sca_dat`, `kappa_dat`, `sca_a_dat`, `sca_b_dat`, `sca_c_dat` and `sca_d_dat` that will be read in the subroutine **`read_para`**. Only the first `NSCAL` real numbers in each of these arrays are used by TURBO and are broadcasted through the arrays `init_sca`, `kappa`, `sca_a`, `sca_b`, `sca_c` and `sca_d` which have the dimensions `(1:NSCAL)`.

NAMELIST FORCING_PARAMETERS AND RELATED VARIABLES

This namelist contains one integer `Fmode` and five real numbers `kinf`, `ksup`, `energy_forcing`, `helicity_para`, `Famp` used to define the forcing. These quantities as well as two additional integers `modes_layer`, `modes_outer_layer` and two other real numbers `inj_energy` and `inj_helicity` will be discussed in the section 2.4.1.

NAMELIST NUMERICS_PARAMETERS AND RELATED VARIABLES

The namelist `numerics_parameters` contains two real numbers `dt_dat`, `cfl` and four integers `fixed_dt`, `time_to_zero`, `dealias`, `userseed` that determine the numerical algorithm used by TURBO. After the namelist `numerics_parameters` declaration, the real number `dt` is also declared.

The first integer fixed_dt determines how the time step is computed. If fixed_dt=1, then the time step dt is fixed and takes the value dt_dat. If fixed_dt=0, the time step varies and is computed automatically using a CFL criterion based on the cfl number.

The integer dealias determines the algorithm used to remove the aliasing error. Two choices are proposed. If dealias=1, a four-step, third order Runge-Kutta method is implemented with phase shifts. If dealias=2 a third-step, third order Runge-Kutta method is implemented with truncation of one third of the modes. This method is usually referred to as the two-third method. Two constant integer parameters are defined after the namelist to simplify the reading of the code:

```
INTEGER, PARAMETER ::  shifts   = 1
INTEGER, PARAMETER ::  twothird = 2
```

The integer time_to_zero determines if the time and the number of files that have been saved (NFIELDS) have to be reset to zero (time_to_zero=1) or if they must be read from file (time_to_zero=0).

The last integer userseed is used to initialize the random numbers. If userseed=0, the random numbers are initialised using date and time, otherwise the value of userseed is used to initialise in a deterministic way the random number generator. The user can then reproduce exactly the same run if needed.

NAMELIST TIME_PARAMETERS

The quantities contained in the namelist time_parameters are the time t, the number of time iteration already done niter_done and the number of times that the fields (velocity, magnetic field, passive scalars...) have been output to files. This namelist is read in the main part of the code and is output by the subroutine **output**.

NAMELIST SAVE_PARAMETERS

The subroutine save_parameters contains six integers. niter_todo represents the total number of iterations desired in the simulation. Each simulation can be split in several runs. This is useful when the queueing system on which the code is running imposes constraints such as time limit. The number of iterations in a run is given by nstep. The integers nsave and nstat represent the number of iterations between two outputs of files and two computations of global statistics respectively. If some of the fields are initialised using random numbers with a prescribed spectrum, the code has the capability to make a number of fake iterations in which the phases of the Fourier mode are allowed to evolve but the amplitudes are kept constant to fit the prescribed spectrum. This number of fake iterations is determined by nbuild. Finally, to avoid too large files when the resolution is very high, TURBO has the capability to output each field in more than one file. This number of file(s) is given by totpart.

It is important to notice that the following namelists must be present in the parameter files : equation_parameters, dim_and_sizes, rot_parameters, numerics_parameters, save_parameters. They will always be read by the subroutine **read_para**. On the contrary, the following namelists are optional : vel_parameters, mag_parameters, forcing_parameters, sca_parameters. They will be read by the subroutine **read_para** only if required by the values of VELEQ, MAGEQ, SCAEQ and forcepara.

The namelist time_parameters is read by the subroutine **read_time** from the files *simuname_t.nnn* where nnn is a three digit file number describing the number of files already saved.

2.2.2 MODULE VARIABLES

The module `variables` is used to declare all the arrays that represent the fields. These array are four-dimensional. Indeed, for the velocity `vel` and the magnetic field `mag`, three indices correspond to the three directions in space, while the last index is used to distinguish the vector components: `vel(qx,qz,qy,i)`. The scalar field `scal` array is also four-dimensional, but the last index corresponds to the species. For each of these variables, one work array with the same structure is also declared in the module. These work arrays are needed by the Runge-Kutta scheme used for time advancement:

```
COMPLEX, ALLOCATABLE, DIMENSION (:,:,:,:)  ::  vel, mag, scal
COMPLEX, ALLOCATABLE, DIMENSION (:,:,:,:)  ::  dvel, dmag, dscal
```

These arrays are allocated in the subroutine **`init_variables`** only if needed, depending of the value of the logicals `VELEQ`, `MAGEQ` and `SCAEQ`.

2.2.3 MODULE FFTWMODULE

This module contains a number of declaration that are needed by the FFTW subroutines. It is part of the distribution of the FFTW library and will not be discussed further here.

2.2.4 MODULE MPI_VAR

This module contains a number of declaration needed when communication subroutines based on MPI are used. First, a number of MPI parameters are defined:

- `ioid`: identification number for the input-output computational node.
- `myid`: identification number for the current computational node.
- `numprocs`: number of computational nodes.
- `ierr`: error code.
- `status_mpi`: mpi communicator

A number of variables needed by FFTW and initialised in the subroutine **`init_fft3d`** are then declared:

- `local_nz`: total number of grid points in the z direction on each computational node.
- `local_z_start`: first index of the grid points in the z direction on each computational node.
- `local_ny_after_transpose`: total number of Fourier modes in the y direction on each computational node.
- `local_y_start_after_transpose`: first index of the Fourier modes in the y direction on each computational node.
- `local_nr`: Total number of Fourier modes on each computational node.

The precision of the simulation depends on the bytes dimensions of the `REAL` and `COMPLEX` (which can be considered as twice the `REAL` type) format. The default size for `REAL` is 4 bytes (single precision) . By using 8 byte `REAL` type, more accurate simulations may be performed but twice more memory is needed (both to store variables in RAM and to save the fields on the hard disk). The computational speed may also be lowered depending on the type of CPU. The code can be compiled either in single or double precision by modifying one line at the top of the file *`makefile`* (`realtype`). The size of the real and complex numbers that have to be used in the communication subroutines is of course also affected and this is taken care of by the following lines that are compiled conditionally depending of this choice.

```
#IF SINGLE
  INTEGER, PARAMETER ::  MPI_MYREAL=MPI_REAL
  INTEGER, PARAMETER ::  MPI_MYCOMPLEX=MPI_COMPLEX
#ENDIF
#IF DOUBLE
  INTEGER, PARAMETER ::  MPI_MYREAL=MPI_DOUBLE_PRECISION
  INTEGER, PARAMETER ::  MPI_MYCOMPLEX=MPI_DOUBLE_COMPLEX
#ENDIF
```

Finally, the module `mpi_var` also declared a number of arrays introduced in 1.3.1 and **??** and allocated and initialised in 2.5.3 are also declared here (`nodey`, `ystart`, `ysize`, `zstart` and `zsize`.

SECTION 2.3

THE SUBROUTINES AND FUNCTIONS IN FILE: TT_FFT.F90

This file contains the Fast Fourier Transform (FFT) subroutines. In the present release of TURBO, the FFT's are based on the FFTW library **??**. However, for portability and simplicity reasons, these FFT's are called using a much simpler syntax and uses only three subroutines. The definitions of these subroutines allow to limit the changes to the code to this *tt_fft.f90* file only if another FFT library should be used.

- **fft3ds**: preforms the transform from real to complex space.
- **ifft3ds**: preforms the transform from complex to real space.
- **init_fft3d**: initiates the FFT subroutines.

2.3.1 FFT3DS

```
CALL fft3ds(freal,fwave)
```

This subroutine transforms a tridimensional array of real numbers into a tridimensional array of Fourier modes using the subroutine **rfftwnd_f77_mpi** from the FFTW library. Several particularities of this FFTW subroutine deserve some explanation. First, it requires a work array of reals that is defined by

```
REAL, ALLOCATABLE, DIMENSION(:)  ::  work
ALLOCATE(work(0:local_nr-1))
```

Second, the subroutine **rfftwnd_f77_mpi** overwrites the input array by the output. This is an non desirable properties in TURBO. For this reason, the input `freal` and the ouput `fwave` are two distinct arrays and the first operation of the subroutine is to store the input into the output (using the proper normalisation parameter C_k):

```
fwave=freal*Ck
```

This is possible because both are defined inside the subroutine **fft3ds** as real arrays:

```
REAL, INTENT(IN)  ::  freal(0:rx,0:ry,0:rz)
REAL, INTENT(OUT) ::  fwave(0:rx,0:ry,0:rz)
```

It is important to notice that this property does not prevent to call the subroutine using a complex array for the output since a complex defined by `complex ::  fwave(0:cx,0:cz,0:cy)` and a real array have exactly the same size. All the calls to **fft3ds** are in fact done with a complex array as the second argument. As mentioned in the section 1.3.2, the order of the last two directions are exchanged in the Fourier transforms (kz is before ky) for efficiency reasons.

2.3.2 IFFT3DS

```
CALL fft3ds(fwave,freal)
```

This subroutine performs the inverse Fourier transform of an array of complex numbers into an array of real numbers. It behaves exactly like **fft3ds**. In particular, both the input and the output arrays are defined internally as real arrays:

```
REAL, INTENT(IN)  ::  fwave(0:rx,0:ry,0:rz)
REAL, INTENT(OUT) ::  freal(0:rx,0:ry,0:rz)
```

but in the calls to **ifft3s**, the input array is systematically an array of complex number. The normalisation is also imposed:

```
freal=fwave*Cr
```

The only minor new feature in **ifft3s** is that the nonphysical last two values in the x direction are set to zero:

```
freal(nx:nx+1,:,:)=0.0
```

2.3.3 INIT_FFT3D

```
CALL init_fft3d()
```

This subroutine computes the number of modes `Ns` and sets the normalization parameters value for the Fourier tansform using the given values for the resolution `nx, ny, nz`.

```
Ns=nx*ny*nz
Ck=1.0
Cr=1.0/Ns
```

The subroutine actually calls two subroutines from the FFTW library to initialize a number of internal arrays and also the local dimensions on each node:

```
cx=nx/2
cy=local_ny_after_transpose-1
cz=nz-1
rx=nx+1
ry=ny-1
rz=local_nz-1
```

TURBO assumes that, on each node, the product of the last two dimensions of real and complex fields are the same. Indeed, the input and the output of both subroutines **`fft3ds`** and **`ifft3ds`** must use the same memory size. There is thus a test that stops the run if `(cy+1)*(cz+1).ne.(ry+1)*(rz+1).`

SECTION 2.4

THE SUBROUTINES AND FUNCTIONS IN FILE: TT_FORCE.F90

This file contains the subroutines responsible used to force the velocity field. Two distinct ways of forcing the fluid are implemented. In the first method, an extra term is added to the right-hand side of the Navier-Stokes equation. It acts as an additional mechanical force. The second method consists in modifying directly the velocity field, usually by maintaining some of its properties (velocity profile, energy in a range of wave vectors) constant. The subroutines present in this file are:

- **`forcing`**: Forcing by adding a mechanical force in the right-hand side of the Navier-Stokes equation.
- **`modify_vel`**: Forcing modifying some velocity properties.

As mentioned in the description of the module `parameters`, `FORCEPARA` can take the following values:

- 0 : No forcing term is added in the Navier-Stokes equation.
- 1 : The forcing injects fixed energy and helicity rates in a shell of wave vectors 1.6.1.
- 2 : The forcing imposes fixed energy and helicity levels in a shell of wave vectors 1.6.2.
- 3 : The forcing is defined by the user in an external subroutine `external_forcing`.
- 4 : Kolmogorov forcing.
- 5 : Imposes a number of constraints on the velocity field defined by the user in the external subroutine **`external_modif_vel`**.

The cases 1, 3 and 4 correspond to a mechanical force and are implemented in the subroutine **`forcing`** 2.4.1. The cases 2 and 5 imply a direct action on the velocity field and are implemented in the subroutine **`modify_vel`** 2.4.2.

2.4.1 FORCING

```
CALL forcing(force)
```

This subroutine computes the force that will be used to update the velocity if `FORCEPARA` takes one of the following values : 1, 3 or 4.

The case `FORCEPARA=1` corresponds to a force that injects a certain amount of energy and of helicity per unit of time (respectively `energy_forcing` and `helicity_para`) in a shell of wave vectors. The shell of wave vectors is defined by a lower value `kinf` and a higher value `ksup`. These four quantities are part of the namelist `force_parameters` define in the module `parameters` and read by the the subroutine **`read_para`**. Details about how to implement such a forcing can be fount in the section 1.6.1. The number of modes affected by this forcing is computed in the subroutine **`init_wavevectors`** 2.5.3 and is stored into the integer `modes_layer`.

The case `FORCEPARA=3` corresponds to a force defined by the user in an external subroutine `external_forcing`.

The case `FORCEPARA=4` corresponds to the Kolmogorov forcing $\vec{f} = A \sin(ky)\ \vec{1}_x$. Since $\sin(ky) = (e^{iky} - e^{-iky})/2$, this forcing is easy to express directly in Fourier space and contain only two modes corresponding to k and $-k$. This is achieved by the following lines of code:

```
DO qy=0,cy
  qqy=qy+local_y_start_after_transpose
  IF (qqy.eq.Fmode) force(0,0,qy,1)=-Famp*I*Ns/2.0
  IF (qqy.eq.ny-Fmode) force(0,0,qy,1)=+Famp*I*Ns/2.0
END DO
```

where `Fmode` is an integer and represents the wave number associated to k and `Famp` corresponds to the forcing amplitude A. Both these parameters are part of the namelist `force_parameters` define in the module `parameters` and read by the the subroutine **`read_para`**.

Regardless of the force type chosen above, the zero divergence condition is imposed on the force at the end of the subroutine.

```
CALL divfree(force)
```

2.4.2 MODIFY_VEL

```
CALL modify_vel(u,delta_energy,delta_helicity)
```

This subroutine modifies the velocity field for the two cases corresponding to `FORCEPARA=2` or 5. Hence, the complex array `u` is both an input as well as an output of the subroutine. This subroutine does not use the module `variables`, so that the velocity has to be explicitly given as the first argument. The other two arguments `delta_energy` and `delta_helicity` are two real numbers in which the variation of energy and the variation of helicity due to the action of the subroutine are measured and stored. This is easily achieved by computing the energy and the helicity before and after the effect of the subroutine.

The case `FORCEPARA=2` corresponds to a modification of the velocity that maintains the level of energy and the level of helicity to constant values prescribed by the real numbers `energy_forcing` and `helicity_para` from the namelist `forcing_parameters` and read by the the subroutine **`read_para`**. Details about how to implement this effect can be found in the section 1.6.2. The number of modes affected by this forcing is computed in the subroutine **`init_wavevectors`** 2.5.3 and is stored into the integer `modes_layer`.

The case `FORCEPARA=5` corresponds to a modification to the velocity field defined by the user in an external subroutine `external_modif_vel`. Typically, the effect of this subroutine could be to maintain a constant profile of velocity in one direction.

SECTION 2.5

THE SUBROUTINES AND FUNCTIONS IN FILE: TT_INIT.F90

This file contains the subroutines needed for the initialization of the variables as well as of a number of arrays needed at various stage in the TURBO code..

- **init_variables**: main subroutine controlling the initialization of variables
- **init_random_seed**: Initialize the random number series
- **init_wavevectors**: initialize the wave vectors and related arrays as well as a number of arrays used to store the data structure.
- **init_vel_field**: initialize the velocity field
- **init_mag_field**: initialize the magnetic field
- **init_sca_field**: initialize the scalar fields
- **init_TaylorGreen**: Taylor Green initial conditions
- **init_random_vec_divfree**: Create a random vectorial field with zero divergence
- **init_random_scal**: create a random scalar field
- **buildphases**: correlates the mode phases while maintaining the mode amplitudes constant.
- **divfree**: Take out the divergence.
- **zero_nhalf**: Put to zero all modes corresponding to `nx/2`, `ny/2` and `nz/2`.
- **symmetrize_all**: Symmetrize all the variables
- **symmetrize_one**: Symmetrize one scalar field in the plane `kx=0`.

2.5.1 INIT_VARIABLES

```
CALL init_variables()
```

This subroutine is called by the main TURBO program to initialize a series of important variables required by the run. First, the fast Fourier transforms and the wave vectors are initialized through calls to two separate subroutines:

```
CALL init_fft3d()
CALL init_wavevectors()
```

The **`init_fft3d()`** initialization subroutine is defined in the *`tt_fft.f90`* file. It only depends on the size of the problem that has to be solved `(nx,ny,nz)`, but not on the type of run (Navier-Stokes, MHD, Quasi-Static, ...).
The **`init_wavevectors()`** subroutine is defined in the *`tt_init.f90`* file. It initializes wave vectors and other related quantities that depend on the numbers of modes `(nx,ny,nz)` as well as on the sizes of the physical domain `(lx,ly,lz)`.

In order to ensure the flexibility of the code, global statistics that are recorded during the simulation are defined externally. Indeed, depending on the problem considered by the user, different statistics can be needed (spectra, one dimensional profiles, two dimensional profiles, ...). They usually required additional variables (arrays) for storage and these arrays must be initialized as well. This is done using the call

```
CALL external_stats_init()
```

where the subroutine **`external_stats_init()`** is defined in the *`tt_exter.f90`* file. In the default distribution, this subroutine is empty and no initialization is actually performed. It is thus the responsibility of the user to declare properly and to initialize the arrays that could be needed in order to collect statistics of the flow.

The velocity, magnetic and scalar fields are initialized independently of each other for flexibility. This allows to initialize and to allocate memory only for the variables that are necessary for the run.

```
IF (VELEQ) CALL init_vel_field()
IF (MAGEQ) CALL init_mag_field()
IF (SCAEQ) CALL init_sca_field()
```

!!Warning!! : *Since only the variables required in the simulation are initialized, any call that would try to use a variable that is not updated by the simulation would directly terminate and crash the run. This could happen in particular when the list of global statistics (computed in the subroutine **`external_stats_compute`**) have not been properly adapted to the simulation. For instance, any attempt to compute the magnetic helicity in a Navier-Stokes run will lead to a crash.*

Finally, the Fourier mode corresponding to `nx/2` or `ny/2` or `nz/2` are set to zero according to the discussion in section 1.4 and the variables are symmetrize to take into account that the original fields are real quantities using the following calls:

```
CALL zero_nhalf()
CALL symmetrize_all()
```

2.5.2 INIT_RANDOM_SEED

```
CALL init_random_seed()
```

The initialization of the random numbers used in various subroutines in TURBO can be done either randomly from the date and time when the parameter `userseed` is set to 0 or in a determinist way using the value of this parameter `userseed`. This choice is made by a fortran case structure. The case `userseed=0` is treated simply using the following fortran call:

```
CALL RANDOM_SEED()
```

The deterministic computation of the seed can be modified by the user by setting the parameter `userseed` to any integer value that will modify the initialization as follows. First, TURBO determines the number of seeds used by the random number generator

```
CALL RANDOM_SEED(SIZE=nseed)
```

Then, it set the values of these seeds and store them:

```
ALLOCATE(seed(nseed))
DO ii=1, nseed
  seed(ii)=ii+userseed+myid
END DO
CALL RANDOM_SEED(PUT=seed)
DEALLOCATE(seed)
```

The use of the formula `seed(ii)=ii+userseed+myid` ensures that a different initialization is used on the different processors and allows the user to change the seeds by modifying the input parameter `userseed`.

2.5.3 INIT_WAVEVECTORS

```
CALL init_wavevectors()
```

The main purpose of this subroutine is to initialize the wavevectors. However, many other important variables are also initialized. These different parts are described in their order of appearance.

First, and this is not really an initialization but rather an information to the user given at an early stage of the run, the precision of the simulation is output. This is done by selecting the size of the type `REAL`. The default is a size of 4 bytes. By using an 8 byte `REAL` type, more accurate simulations may be performed but twice more memory is needed (both to store variables in RAM and to save the fields on the hard disk). The computational speed may also be lowered depending on the type of CPU.

```
IF (myid.eq.ioid) THEN
  SELECT CASE (SIZEOF_REAL)
  CASE(4)
    WRITE(6,*) 'This is a single precision run'
  CASE(8)
    WRITE(6,*) 'This is a double precision run'
  END SELECT
END IF
```

Next, a number of arrays related to the way data are distributed amongst the processors are also initialized.

- `ystart(0:numprocs-1)` contains the values of `local_y_start_after_transpose`
- `ysize(0:numprocs-1)` contains the values of `local_ny_after_transpose`
- `zstart(0:numprocs-1)` contains the values of `local_z_start`
- `zsize(0:numprocs-1)` contains the values of `local_nz`

This allows all to have a direct access to the complete information on the data stucture from each processor. Two closely related arrays, `whereis_z(0:nz-1,2)` and `whereis_qy(0:ny-1,2)` introduced respectively in the sections 1.3.1 and 1.3.2 are then defined.

Next, a number of arrays related to the wavevectors are allocated and initialized. The related code lines are not reproduced in details here. The wavevectors are defined as real one dimensional arrays `kx(0:cx), ky(0:cy), kz(0:cz)` and are initialized according to the rules given in section 1.3.2. For simplicity in the writing of the evolution equation and, especially, for expressing derivatives, complex arrays `ikx(0:cx), iky(0:cy), ikz(0:cz)` are also defined and initialized as `ikx=I*kx`, `iky=I*ky` and `ikz=I*kz`. Also, a one dimensional array `MM(0:cx)` that appears in the expression of the Parseval theorem is also initialized. Four three-dimensional arrays are also defined and initialized : `knorm` and `ksquare` contains respectively the norm and the square of the norm of the wavevector, `km2` contains the inverse of `ksquare` corrected to avoid a singularity for $\vec{k} = 0$ (`km2`($\vec{k} = 0$) is set to 1/`eps`) and `zeros_twothird` is used for the de-aliasing 2/3 method. The smallest distance between two grid point, `dx=lx/nx`, `dy=ly/ny`, `dz=lz/nz` are computed and if rotation is present, the `rot_omega` is initialized. The smallest wavevector in each direction, `k0x=`2π`/lx`, `k0y=`2π`/ly`, `k0z=`2π`/lz` (the largest of these three values is retained to define `k0`) and the largest wave vector in each direction `kMx=`π`nx/lx`, `kMy=`π`ny/ly`, `kMz=`π`nz/lz` (the smallest of these three values is retained to define `kM`) are also computed. For the 2/3 de-aliasing method, the largest wave vectors are computed using the

kM from the smallest grid (2/3 of the given numeric grid) since only the modes defined on this grid have physical meaning. Finally, at the end of the subroutine, the number of modes that are included in the forcing layer defined by `kinf` $< |\vec{k}| <$ `ksup` is computed and output with a number of parameters related to the forcing.

2.5.4 INIT_VEL_FIELD

```
CALL init_vel_field()
```

This subroutine initializes the velocity field. First memory is allocated to the velocity array and another array with the same dimension used in the time-stepping procedure.

```
ALLOCATE(vel(0:cx,0:cz,0:cy,3))
ALLOCATE(dvel(0:cx,0:cz,0:cy,3))
```

Depending on the `init_vel` parameter value given in the parameter .par file, we have:

- `init_vel= restart`: The initial velocity field of the run is read from a file using the subroutine **`read_vec_divfree`** (see 2.6.4-2.6.5)
- `init_vel=random`: The initial velocity field is generated using random numbers in the subroutine **`init_random_vec_divfree`**.
- `init_vel=taylor_green`: The initial field is defined as a Taylor-Green vortex in the subroutine **`init_Taylor_Green`**.
- `init_vel=extern`: The initial field is defined in an external subroutine **`external_init_vel`** provided by the user in the file *`tt_exter.f90`*.

After the field has been read or generated, the zero divergence condition is enforced

```
CALL divfree(vel)
```

2.5.5 INIT_MAG_FIELD

```
CALL init_mag_field()
```

This subroutine initializes the magnetic field. First memory is allocated to the velocity array and another array with the same dimension used in the time-stepping procedure.

```
ALLOCATE(mag(0:cx,0:cz,0:cy,3))
ALLOCATE(dmag(0:cx,0:cz,0:cy,3))
```

Depending on the `init_mag` parameter value given in the parameter .par file, we have:

- `init_mag=0`: Read existing file
- `init_mag=1`: Initialize as a random field
- `init_mag=2`: Initialize using tha Taylor-Green vortex
- `init_mag=3`: Initialize using a definition provided by the user externally

After the field has been read or generated, the zero divergence condition is enforced

```
CALL divfree(mag)
```

2.5.6 INIT_SCA_FIELD

```
CALL init_sca_field()
```

This subroutine initializes the scalar fields. First memory is allocated to the scalar collection of NSCAL fields and other arrays with the same dimension used in the time-stepping procedure.

```
ALLOCATE(sca(0:cx,0:cz,0:cy,NSCAL))
ALLOCATE(sca(0:cx,0:cz,0:cy,NSCAL))
```

For each NSCAL, depending on the init_sca_dat parameter value given in the parameter .par file, we have:

- init_sca_dat=0: Read existing file
- init_sca_dat=1: Initialize as a random field
- init_sca_dat=2: Initialize using a definition provided by the user externally

2.5.7 INIT_TAYLORGREEN

```
CALL init_TaylorGreen(u, angle, amplitude)
```

This subroutine initializes the complex vector field `u` as a Taylor Green vortex. Itdepends on two parameters: an amplitude and an angleand is defined in real space as follows:

$$
\begin{aligned}
u_x(x,y,z) &= \frac{2\,\texttt{amplitude}}{\sqrt{3}}\,\sin(\texttt{angle}+\frac{2\pi}{3})\sin(x)\cos(y)\cos(z)\\
u_y(x,y,z) &= \frac{2\,\texttt{amplitude}}{\sqrt{3}}\,\sin(\texttt{angle}-\frac{2\pi}{3})\cos(x)\sin(y)\cos(z)\\
u_z(x,y,z) &= \frac{2\,\texttt{amplitude}}{\sqrt{3}}\,\sin(\texttt{angle})\cos(x)\cos(y)\sin(z)
\end{aligned}
$$

In Navier-Stokes turbulence, when the velocity field is initialized as a Taylor Green vortex, it becomes turbulent and the entire energy spectrum is rapidely filled [**?**]. This subroutine can be used also to initialize the magnetic field. The field components are computed using formula (1.21):

```
DO ix=0, rx
  DO iy=0, ry
    DO iz= 0, rz
      ux(ix,iy,iz)=SIN(ix*lx/nx)*COS(iy*ly/ny)*COS((iz+local_z_start)*lz/nz)
      uy(ix,iy,iz)=COS(ix*lx/nx)*SIN(iy*ly/ny)*COS((iz+local_z_start)*lz/nz)
      uz(ix,iy,iz)=COS(ix*lx/nx)*COS(iy*ly/ny)*SIN((iz+local_z_start)*lz/nz)
    END DO
  END DO
END DO
ux=ux*2.0/SQRT(3.0)*amplitude*SIN(angle+twopi/3.0)
uy=uy*2.0/SQRT(3.0)*amplitude*SIN(angle-twopi/3.0)
uz=uz*2.0/SQRT(3.0)*amplitude*SIN(angle)
```

As mentioned in the section 1.3.1, in the present distribution of TURBO, the Fourier transforms are obtained using the library FFTW. This library requires that `rx=nx+1`. The unused grid points corresponding to `nx` and `nx+1` are set to zero.

```
ux(nx:nx+1,:,:)=0.0
uy(nx:nx+1,:,:)=0.0
uz(nx:nx+1,:,:)=0.0
```

Finally the field is Fourier transformed to complex space:

```
CALL fft3ds(ux,uc)
u(:,:,:,1)=uc
CALL fft3ds(uy,uc)
u(:,:,:,2)=uc
CALL fft3ds(uz,uc)
u(:,:,:,3)=uc
```

2.5.8 INIT_RANDOM_VEC_DIVFREE

```
CALL init_random_vec_divfree(u,pa,pb,pc,pd)
```

This subroutine generates a random vectorial field `u` with zero divergence using up to four parameters `pa`, `pb`, `pc` and `pd` to define its energy spectrum. The output is the vector field in complex space.

The subroutine is build to ensure that the same random numbers are generated independtly of the number of processors used in the computation. Indeed, the random numbers are computed in a determinist manner based on i) seeds fully determined by the subroutine **`init_random_seed`** and ii) the number of random numbers that have already been generated. The seeds are computed in a way that is independent of the number of processors, `numprocs`. It is thus sufficient to ensure that the same number of random numbers have already been computed when the random field is generated on the processor `myid`.

This is achieved in noting two important properties: 1) as explained below, each vector mode `u(qx,qy,qz,1:3)` is constructed using 3 random numbers (ϕ, θ_1 and θ_2) and 2) for each value of `qy` there are `nx*nz/2` vector modes. Hence, each processor will initialize `ysize(myid)*nx*nz/2` vector modes that correspond to `qy` between `ystart(myid)` and `ystart(myid)+ysize(myid)-1`.

The subroutnie **`init_random_vec_divfree`** is thus organised in three steps. First, on each processor, `3*nx*nz/2*ystart(myid)` random numbers are generated but not used. Second `3*nx*nz/2*ysize(myid)=3*nx*nz/2*cy` random numbers are used to build the vector modes with `qy` between `ystart(myid)` and `ystart(myid)+ysize(myid)-1`. Third, `3*nx*nz/2*(ny-ystart(myid)-ysize(myid))` random numbers are generated but not used. This last step ensures that on each processor, when leaving the subroutine, the number of random numbers that have been generated by the call is exactly `3*nx*nz/2*ny`, i.e. a value independent of `numprocs`.

The first step is achieved using the following lines of code:

```
IF (myid.ne.ioid) THEN
  DO qx=0,3*nx*nz/2*ystart(myid)-1
    CALL RANDOM_NUMBER(theta1)
  END DO
END IF
```

The second step requires more algebra. The amplitude of the vector modes is computed making use of the function **`external_energy_spectrum(k,pa,pb,pc,pd)`** defined in the *`tt_exter.f90`* file.

$$A = \left(\frac{\texttt{external\_energy\_spectrum}}{2\pi k^2} \right)^{1/2}$$

The vector modes are defined using the following formula:

$$\begin{aligned} u_x &= +\alpha k k_y + \beta \frac{k_x k_z}{k k_{xy}} \\ u_y &= -\alpha k k_x + \beta \frac{k_y k_z}{k k_{xy}} \\ u_z &= -\beta \frac{k_{xy}}{k} \end{aligned} \tag{2.1}$$

where α and β are the following functions of the random numbers θ_1, θ_2 and ϕ:

$$\begin{aligned} \alpha &= A\cos(2\pi\phi)[\cos(2\pi\theta_1) + i\sin(2\pi\theta_1)] \\ \beta &= A\sin(2\pi\phi)[\cos(2\pi\theta_2) + i\sin(2\pi\theta_2)] \end{aligned}$$

and where

$$k_{xy} = \left(k_x^2 + k_y^2\right)^{1/2}$$

It is easy to show that the formula 2.1 ensures that $\vec{k}.\vec{u}(\vec{k}) = 0$ independently of α and β. For $k_x = 0$, $k_y = 0$ and therefore $k_{xy} = 0$, we can not use the formula 2.1. In that case, the vector mode is computed as follows:

$$\begin{aligned} u_x &= \frac{1}{\sqrt{2}}(\alpha + \beta) \\ u_y &= \frac{1}{\sqrt{2}}(-\alpha + \beta) \\ u_z &= 0 \end{aligned}$$

The third step is achieved using:

```
IF((ny-ystart(myid)-ysize(myid)).ne.0)THEN
  DO qx=0,3*nx*nz/2*(ny-ystart(myid)-ysize(myid))-1
    CALL RANDOM_NUMBER(theta1)
  END DO
END IF
```

2.5.9 INIT_RANDOM_SCAL

```
CALL init_random_scal(sc,pa,pb,pc,pd)
```

This subroutine generates a random scalar field `sc` using up to four parameters `pa`, `pb`, `pc` and `pd` to define its energy spectrum. The output is the scalar field in complex space. As **`init_random_vec_divfree`**, the subroutine **`init_random_scal`** is build to ensure that the same random numbers are generated independtly of the number of processors used in the computation. It is organised in three steps, very similar to those used in **`init_random_vec_divfree`**, except that only one random number (instead of 3) is needed for each mode. We only discuss the second step, in which the amplitude of the modes is computed making use of the **`external_energy_spectrum(k,pa,pb,pc,pd)`** function defined in the *`tt_exter.f90`* file

$$A = \left(\frac{\texttt{external\_energy\_spectrum}}{2\pi k^2} \right)^{1/2} \tag{2.2}$$

The scalar field is defined with random phases θ:

$$s = Ae^{i\theta} \tag{2.3}$$

Here again, starting with the `qy` loop is important to guarantee equivalence between runs using different number of nodes:

```
DO qy=0,cy
  DO qx=0,cx
    DO qz=0,cz
      CALL RANDOM_NUMBER(theta)
      theta = twopi*theta
      k = MAX(knorm(qx,qz,qy),EPS)
      amp = SQRT(external_energy_spectrum(k,pa,pb,pc,pd)/2.0/pi/k**2)
      sc(qx,qz,qy)=amp*EXP(I*theta)
    END DO
  END DO
END DO
```

2.5.10 BUILDPHASES

```
CALL buildphases()
```

This subroutine is called only from the subroutine **init_variables** when the user provided parameter nbuild is larger than 0. In that case, nbuild time steps are used in which all the fields that are initialized randomly are advanced in time, using:

```
CALL compute_dt()
CALL update_variables(iter)
```

and then rescaled to the spectrum prescibed by the user. For instance, if the velocity is computed (VELEQ is .true.) and initialized using random numbers, then it is rescaled as follows:

```
DO qy=0,cy
  DO qz=0,cz
    DO qx=0,cx
      k = MAX(knorm(qx,qz,qy),EPS)
      amp1 = SQRT(external_energy_spectrum(k,vel_a,vel_b,vel_c,vel_d)/2.0/pi/k**2)
      amp2 = ABS(vel(qx,qz,qy,1))**2+ABS(vel(qx,qz,qy,2))**2+ABS(vel(qx,qz,qy,3))**2
      fac = amp1/SQRT(MAX(amp2,EPS))
      vel(qx,qz,qy,1:3) = vel(qx,qz,qy,1:3)*fac
    END DO
  END DO
END DO
```

The fields that are not initialized as random fields are simply re-initialized to their initial value. The purpose of this procedure is to build-up the phases of the randomly initialized field so that they do correspond as much as possible to real turbulence. Indeed, the subroutines **init_random_vec_divfree** and **init_random_scal** are used to prescribe the amplitudes of the velocity, magnetic or scalar Fourier modes but the phases of these modes are totally randomly distributed. Actual measurements of these phases show however that they are not necessarily uniformly distributed. Since it is very difficult to impose directly a possible phase correlation, using the time integration to produce realistic phases, while keeping the spectrum constant to impose the Fourier mode amplitudes, is a convenient way to produce more realistic turbulent fields.

During each of these nbuild time steps, the code alos computes and outputs several global statistics by calling the subroutine **external_stats_compute**. It is important to notice that quantities such as the energy, the dissipation should be unchanged by the rescaling procedure, while kinetic helicity and magnetic helicity are expected to be modified when the phases are changed.

2.5.11 DIVFREE

```
CALL divfree(a)
```

This subroutine removes the divergence of a vector field a, expressed in the complex space. It modifies the input vector `a` by means of the projector operator $P_{ij}(\vec{k}) \equiv (\delta_{ij} - k_i k_j / k^2)$. If we denote the input vector field as a_i^{in} and the output divergence free field as $a_i^{\text{out}}(\vec{k})$ then we have:

$$\begin{aligned} a_i^{div\ free}(\vec{k}) &\equiv P_{ij}(\vec{k}) a_j^{\text{in}}(\vec{k}) = (\delta_{ij} - \frac{k_i k_j}{k^2}) a_j^{\text{in}}(\vec{k}) \\ &= a_i(\vec{k} - k_i \nabla \cdot \vec{a}^{\text{in}})\, k^{-2} \end{aligned} \tag{2.4}$$

This is easily achieved using the following lines:

```
DO qy=0, cy
  k2=ky(qy)
  DO qz= 0, cz
    k3=kz(qz)
    DO qx=0, cx
      k1=kx(qx)
      div=(k1*a(qx,qz,qy,1)+k2*a(qx,qz,qy,2)+k3*a(qx,qz,qy,3))*km2(qx,qz,qy)
      a(qx,qz,qy,1)=a(qx,qz,qy,1)-k1*div
      a(qx,qz,qy,2)=a(qx,qz,qy,2)-k2*div
      a(qx,qz,qy,3)=a(qx,qz,qy,3)-k3*div
    END DO
  END DO
END DO
```

2.5.12 ZERO_NHALF

```
CALL zero_nhalf()
```

According to the discussion in section 1.4, the Fourier modes corresponding to `nx/2` or `ny/2` or `nz/2` have to be set to zero. As far as the modes with `nx/2` and `nz/2` are concerned, this is easy to achieve. For instance, if we consider the velocity field, these modes are set to zero as follows:

```
vel(nx/2,:,:,:)=0
vel(:,nz/2,:,:)=0
```

The modes corresponding to `ny/2` have to be treated a bit more carefully since we have to determine which processor is in charge of theses modes. However, this information is easily derived from the tensor `whereis_ky`:

```
proc_id = whereis_ky(ny/2,1)
qqy     = whereis_ky(ny/2,2)
```

where `proc_id` gives the processor id where modes `ny/2` are stored and `qqy` gives the location of theses modes on this processor. The modes `ny/2` are then set to zero as follows:

```
IF (myid.eq.proc_id) THEN
  vel(:,:,qqy,:)  =0
END IF
```

!!Warning!! : *The subroutine* ***zero_nhalf*** *also sets to zero the modes of the work arrays* `dvel`, `dmag` *and* `dscal` *depending on the type of simulation. This subroutine can thus only be called once these work arrays have been properly allocated.*

2.5.13 SYMMETRIZE_ALL

```
CALL symmetrize_all()
```

Subroutine used to symmetrize all the variables depending on the problem (Nabier-Stokes, MHD, ...). See the subroutine **symmetrize_one** below for details.

2.5.14 SYMMETRIZE_ONE

```
CALL symmetrize_one(s)
```

In most of the modern FFT libraries, the Fourier transform of a real array is stored as a complex array `s` with half the dimension in one direction in order to take into account the property 1.11. In the case of the FFTW used in TURBO, the first direction has dimension `nx` in real space and `nx/2` in Fourier space and the transform automatically assumes that the modes with negative component of the wave vector, `kx`, are the complex conjugate of the modes with the opposite (and thus positive) `kx`. However, when initializing an array in Fourier space, a number of additional conditions on the modes with `kx=0` have to be imposed:

$$\tilde{s}^d(0,k_z,k_y)^* = \tilde{s}^d(0,-k_z,-k_y)\,. \tag{2.5}$$

These additional conditions are split into two part.

A) First, for stricly positive `ky`, the index `qy` is determined and is stored into `qy1` while the index corresponding to `-ky` is `ny-qy` and is stored inot `qy2`. Simultaneously, the id of the processors that store the modes corresponding to `qy1` and `qy2` as well as the local index of these modes are determined:

```
qy1=qy
qy2=ny-qy
pid1=whereis_ky(qy1,1)
pid2=whereis_ky(qy2,1)
qqy1=whereis_ky(qy1,2)
qqy2=whereis_ky(qy2,2)
```

If `pid1` is different from `pid2`, the one-dimensional array `s(0,:,qqy1)` is sent from processor `pid1` to processor `pid2` and is stored in the array `sb`

```
IF (myid.eq.pid1) CALL MPI_SEND(s(0,:,qqy1),nz,MPI_MYCOMPLEX,pid2,10,MPI_COMM_WORLD,ierr)
IF (myid.eq.pid2) CALL MPI_RECV(sb ,nz,MPI_MYCOMPLEX,pid1,10,MPI_COMM_WORLD,status_mpi,ierr)
```

Otherwise (`pid1=pid2`), sb is computed locally:

```
IF (myid.eq.pid1) sb=s(0,:,qqy1)
```

Then, on the node `pid2` the relation 2.5 is imposed :

```
IF (myid.eq.pid2) s(0,:,qqy2)=CONJG(sb)
```

B) Second, for `ky=0`, the corresponding processor and local index are first determined:

```
pid1=whereis_ky(0,1)
qqy1=whereis_ky(0,2)
```

Then, the condition $\tilde{a}^d(0,k_z,0)^* = \tilde{a}^d(0,-k_z,0)$ is imposed:

```
IF (myid.eq.pid1) THEN
  DO qz=1,nz/2-1
    s(0,nz-qz,qqy1) = CONJG(s(0,qz,qqy1))
  END DO
END IF
```

SECTION 2.6

THE SUBROUTINES AND FUNCTIONS IN FILE: TT_IO.F90

This file contains the subroutines responsible for the input and the output for TURBO. To understand these subroutines, it is recommended that the reader familiarizes himself with the structure of the parameter file (section **??**) and the way the simulation is resumed (section **??**).
The subroutines contained in this file and their main function are:

- **output**: the main output subroutine
- **write_vec_divfree**: writes divergent free vector fields to disk
- **read_vec_divfree**: reads divergent free vector fields from disk
- **write_scal**: writes scalar fields to disk
- **read_scal**: reads scalar fields from disk
- **read_time**: reads the time and restart parameters
- **read_para**: reads the parameter files

2.6.1 FILE FORMAT

The velocity and magnetic field vectors as well as the scalar fields are saved in direct acces files as a collection of records, corresponding to slices of Fourier modes. A slice is defined as a two-dimensional complex array `slice(0:cx,0:cz)`. Considering the Fourier space representation 1.3.2 of fields in TURBO, all the Fourier modes corresponding to a slice are always handled by a unique processor. Typical slices are `vel(:,:,qy,i)` or `scal(:,:,qy,iscal)`. The length of a slice is independent of the node index (`myid`) and is determined by

```
reclength=2*(cx+1)*(cz+1)*SIZEOF_REAL
```

The files are opened using the following command lines:

```
OPEN (UNIT=10, FILE=filename, STATUS='UNKNOWN', ACTION='WRITE',ACCESS='DIRECT',&
  FORM='UNFORMATTED', RecL=reclength)
```

2.6.2 INPUT/OUPUT NODE

The **mpi_var** module defines the parameter `ioid=0`. It is used to determine the value of the node index from which all the "write to" and "read from" file operations are performed. A subroutine **transfer_slice** has been defined to send the data in a slice from one node to another. In practice, it is used to collect the slices from all nodes to the `ioid` node in the subroutines **write_vec_divfree** and **write_scal**, while is it used to send the slices from the node `ioid` to the appropriate nodes in the subroutines **read_vec_divfree** and **read_scal**.

2.6.3 OUTPUT

```
CALL output(save_stat)
```

This simple subroutine is responsible for the saving the fields at the current time of the simulations. The fields that are output depend of course on the simulation type. Velocity and magnetic fields are saved to files using the subroutine **write_vec_divfree**, while the scalar fields are saved using the subroutine **wirte_scal**. Both subroutines are described below.

It also output two files with information on the current time and the index of the last saved field. This index is stored in the integer `nfields` and is transformed into an array of 3 characters `cnum` using

```
WRITE(cnum,'(i3.3)') nfields
```

Hence, if `nfields=12` the value stored in the `cnum` is 012. If the simulation name is "mysimu", the namelist `time_parameters` is then saved into the file *mysimu_t_012*. Simulataneously, the code saves the value of `nfield` into the file *mysimu.n*.

Intermediate statistics files are then saved if the integer argument `save_stat=1` using the following calls:

```
CALL external_stats_write()
CALL external_stats_end()
CALL external_stats_init()
```

These statistics are intended to be running averages of a number of relevant quantities. Obviously, these quantities will depend on both the type of simulation as well as of the specific interest of the user. For these reasons, they are computed using external subroutine from the file *xx_exter.f90*. The first call obviously save the intermediate statistics. The second call is intended to stop the computation of running average. The third one, re-initialize the arrays needed in the computation of the intermediate statistics.

Finally, the subroutine updates the value of the parameter `nfields`.

2.6.4 WRITE_VEC_DIVFREE

```
CALL write_vec_divfree(u,beginname)
```

This subroutine saves the Fourier space representation of a divergence free vector field `u`, typically a velocity or a magnetic field, into one file or several files (see also 2.6.1 and 2.6.2). The field is declared by:

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy,3), INTENT(IN) ::  u
```

The number of files used to save the vector `u` in several files is determined by the parameter `totpart`. Splitting the files is useful when very high resolutions are used. The ratio `ny/totpart` must be an integer otherwise the run stops. The names of the files depend on the number of previously stored fields through the value of `nfields`. For example, assuming that the simulation name is "mysimu" and considering `nfields=12` and `totpart=2`, the velocity field will be stored in two files corresponding to two values of the parameter `ipart` (0 and 1). The file names are then *mysimu_u00.012* and *mysimu_u01.012*. These names are built using the commands:

```
WRITE(cnum,'(i3.3)') nfields
WRITE(partnum,'(i2.2)') ipart
filename=beginname//partnum//'.'//cnum
```

The input and output subroutines in TURBO take advantage of the divergence free condition to save only two components of the vector in Fourier space, typically the component `u(:,:,:,1)` and `u(:,:,:,3)`. Indeed, the last component can be reconstructed using the divergence free condition:

$$k_j \tilde{u}_j \equiv k_x \tilde{u}_x + k_y \tilde{u}_y + k_z \tilde{u}_z = 0 \Rightarrow \tilde{u}_y = \frac{-(k_x \tilde{u}_x + k_z \tilde{u}_z)}{k_y} \tag{2.6}$$

However, the component with $k_y = 0$ cannot reconstructed using this formula and the subroutine also saves the modes corresponding to the slice `u(:,:,0,1)` at the beginning of the first file using the following commands:

```
IF (ipart.eq.0) THEN
  pid=whereis_ky(0,1)
  qqy=whereis_ky(0,2)
  CALL transfer_slice(u(:,:,qqy,2),pid,utempx,ioid)
  IF (myid.eq.ioid) THEN
    WRITE(10,rec=reccount) utempx
    reccount=2
  END IF
END IF
```

where the array `where_is` is used to find on which node (`pid`) and at which location (`qqy`) the slice corresponding to $k_y = 0$ is stored. The final part of the subroutine is a loop on `qy` in which the slices `u(:,:,qy,1)` and `u(:,:,qy,3)` are sent to the node `ioid` and then written to the appropriate file.

The number of slices, or equivalently the number of records, saved in the first file (`ipart=0`) generated by the subroutine **`write_vec_divfree`** is thus `2 ny / totpart + 1`, while the other files store `2 ny / totpart` slices. If only one part is used (`totpart=1`), the vector fields are saved on disk using a unique direct access file with `2 ny + 1` records, each of them corresponding to a slice. The first slice correspond to $k_y = 0$ and component y, then $k_y = k_y^0$ for the component x followed by the component z, then $k_y = 2\ k_y^0$ for the component x followed by the component z and so on...

2.6.5 READ_VEC_DIVFREE

```
CALL read_vec_divfree(u,beginname)
```

This subroutine is very similar to the subroutine **write_vec_divfree** except that it reads from file and dispatch the information to the appropriate nodes instead of gathering inofrmation from the nodes and save to file. The structures of both subroutines are thus very similar.

Once the files have been read, the y component of the vector is reconstructed (except for $k_y = 0$) using the following commands:

```
DO qy=0,cy
  IF (ABS(ky(qy)).gt.EPS) THEN
    DO qz=0, cz
      DO qx=0, cx
        u(qx,qz,qy,2)=-(u(qx,qz,qy,1)*kx(qx)+u(qx,qz,qy,3)*kz(qz))/ky(qy)
      END DO
    END DO
  END IF
END DO
```

2.6.6 WRITE_SCAL

```
CALL write_scal(s,beginname)
```

This subroutine is used to write a scalar field to file. The subroutine structure is very similar to the one described in the subroutine **write_vec_divfree**. The subroutine **write_scal** could actually be used to output the components of a vector individually if needed for postprocesing.

The name of the file is a little bit different. Indeed, considering the scalar field with `iscal=3`, and assuming that the simulation name is "mysimu" and again that `nfields=12` and `totpart=2`, the scalar field will be stored in two files corresponding : *mysimu_s03.00.012* and *mysimu_s03.01.012*. These names are built using the commands:

```
CHARACTER(LEN=2) ::  partnum
WRITE(partnum,'(i2.2)') ipart
filename=beginname//partnum//'.'//cnum
```

2.6.7 READ_SCAL

```
CALL read_scal(s,beginname)
```

This subroutine is very similar to the subroutine **write_scal** except that it reads from file and dispatch the information to the appropriate nodes instead of gathering inofrmation from the nodes and save to file. The structures of both subroutines are thus very similar.

2.6.8 READ_TIME

```
CALL read_time()
```

The subroutine initalizes the parameter stored in the namelist `time_parameters`. Its action depends on the input parameter `time_to_zero`. If `time_to_zero=1`, the values of the parameters in the namelist `time_parameters` are set to their default value (`t=0`, `niter_done=0` and `nfields=0`).

The fisrt parameter `t` is obviously the time. It is a real number. The second parameter `niter_done` is the total number of iterations that have already been performed in previous runs of the simulation. This parameter is useful for long simulation that have to be split into several runs for computer queue constraints. The third parameter, `nfields`, refers to the number of files that have already saved. Indeed, the subroutine **`write_vec_divfree`** and **`write_scal`** are used to save on files snapshots of the variables in the course of the simulation. Each time these files are saved, `nfield` is incremented by 1, its value is stored into the file *`simname.n`* and a file *`simname_t.cnum`* is created with the current value of the namelist `time_parameters` (see 2.6.3).

If `time_to_zero=0`, then the value of `nfields` is read from the file *`simname.n`* and the parameters in the namelist `time_parameters` are read from the file *`simname_t.cnum`*

The subroutine also broadcasts the three parameters to all computational nodes:

```
CALL MPI_BCAST(niter_done,1,MPI_INTEGER,ioid,MPI_COMM_WORLD,ierr)
CALL MPI_BCAST(nfields,1,MPI_INTEGER,ioid,MPI_COMM_WORLD,ierr)
CALL MPI_BCAST(t,1,MPI_MYREAL,ioid,MPI_COMM_WORLD,ierr)
```

2.6.9 READ_PARA

```
CALL read_para()
```

This subroutine is responsible for reading the parameter of the simulation. It is called at the beginning of the main program. Though this subroutine is quite long, it is rather simple and mainly reads a series of namelists and broadcasts the parameters to all the computational nodes. Some information on the type of simulation is output on screen. A few parameters are initialized directly as their use will be required immediately such as the logical parameters. These parameters are:

- `VELEQ`: its value is set to `.TRUE.` if the velocity field is computed, `.FALSE.` if not.
- `MAGEQ`: its value is set to `.TRUE.` if the magnetic field is computed, `.FALSE.` if not.
- `SCAEQ`: its value is set to `.TRUE.` if the scalar field is computed, `.FALSE.` if not.
- `ROTEQ`: its value is set to `.TRUE.` if rigid rotation is imposed, `.FALSE.` if not.

The above parameters allow the code to be flexible and only allocate memory or perform computation if needed. For instance if `MAGEQ=.FALSE.`, no magnetic field equation will be solved. As a consequence, no memory will be allocated to the magnetic related arrays, no computation performed using the magnetic field and even the parameters list related to the magnetic field `mag_parameters` will not be read.

The namelists that are read (if needed) are listed here:

- `equation_parameters`
- `vel_parameters`
- `mag_parameters`
- `sca_parameters`
- `dim_and_sizes`
- `forcing_parameters`
- `numerics_parameters`
- `save_parameters`

To understand better the namelist refer to the section **??**.

2.6.10 TRANSFER_SLICE

```
CALL transfer_slice(slice,fromid,destination,destid)
```

This subroutine simply send the information stored in the variable `slice` on the node `fromid` to the variable `destination` on the node `destid`. Both `slice` and `destination` are supposed to be complex arrays of dimensions `(0:cx,0:cz)`. The information is sent by using the MPI commands "MPI_SEND" and "MPI_RECV" if the node indices `fromid` and `destid` are different. Otherwise, a simple equality is used:

```
IF (destid.ne.fromid) THEN
  IF (myid.eq.fromid) CALL MPI_SEND(slice,(cx+1)*(cz+1),
       MPI_MYCOMPLEX,destid,10,MPI_COMM_WORLD,ierr)
  IF (myid.eq.destid) CALL MPI_RECV(destination,(cx+1)*(cz+1),
       MPI_MYCOMPLEX,fromid,10,MPI_COMM_WORLD,status_mpi,ierr)
ELSE
  IF (myid.eq.destid) destination=slice
END IF
```

SECTION 2.7

THE SUBROUTINES IN THE FILE: TT_STAT.F90

This file contains the subroutines that compute a number of global statistics. The subroutines contained in this file and their main function are:

- **correl_vector**: computes the correlation between two vectors quantities.
- **correl_scalar**: computes the correlation between two scalar quantities.
- **correl_helicity**: computes the correlation between a vector and the curl of another vector.
- **compute_diss_QS**: computes the quasi static dissipation.
- **compute_energy_real**: computes the energy in real space.
- **compute_Rlambda**: computes R_λ.
- **compute_kmaxeta**: computes $k_{\max}\,\eta$.

Most of these subroutines (2.7.1, 2.7.2, 2.7.3, 2.7.4 and 2.7.5) compute sums over very large number of terms in order to approximate some integrals. If they are computed in a single precision run, the result might depend on the number of nodes used for the computation. A better accuracy is obtained by computing internally (i.e. inside these subroutines) the sums using double precision, independently of the type of run chosen (single or double precision). This has been implemented.

2.7.1 CORREL_VECTOR

```
CALL correl_vector(a1,a2,j,factor,result)
```

The input variables and input parameters are defined as

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy,3), INTENT(IN) ::  a1, a2
INTEGER, INTENT(IN) ::  j
REAL, INTENT(IN)    ::  factor
```

This subroutine computes the following quantity C (ℓ is assumed to be integer):

$$C = \frac{\texttt{factor}}{V} \int d\vec{x} \ \left(\Delta^{j/2}\vec{a}_1(\vec{x})\right) \cdot \vec{a}_2(\vec{x}) \tag{2.7}$$

$$= \frac{\texttt{factor}}{V} \int d\vec{x} \ \left(\Delta^{j/2-\ell}\vec{a}_1(\vec{x})\right) \cdot \left(\Delta^{\ell}\vec{a}_2(\vec{x})\right) \tag{2.8}$$

$$= \frac{\texttt{factor}}{V} \int d\vec{x} \ \vec{a}_1(\vec{x}) \cdot \left(\Delta^{j/2}\vec{a}_2(\vec{x})\right) \tag{2.9}$$

and stores it into the real number `result`. The equality between the above expressions is valid because TURBO is restricted to problems with periodic boundary conditions (see section 1.5 for details on how this type of integrals can be computed using TURBO). Examples of quantities that can be computed with this subroutine are:

- The kinetic energy per unit of volume e_k ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = 0$, $\texttt{factor} = 0.5$).
- The energy injection rate per unit of volume ϵ_{inj} due to an external forcing $\vec{f}$ ($\vec{a}_1 = \vec{u}$, $\vec{a}_2 = \vec{f}$, $j = 0$, $\texttt{factor} = 1.0$).
- The enstrophy per unit of volume Ω ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = 2$, $\texttt{factor} = -0.5$).
- The kinetic energy dissipation ϵ_u ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = 2$, $\texttt{factor} = \nu$).
- The palinstrophy per unit of volume $\mathcal{P}$ ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = 4$, $\texttt{factor} = 0.5$). Note that the palinstrophy multiplied by -2ν gives the dissipation of enstrophy per unit of volume.
- The cross-helicity per unit of volume h_c ($\vec{a}_1 = \vec{u}$, $\vec{a}_2 = \vec{b}$, $j = 0$, $\texttt{factor} = 1.0$).
- The integral length L_{int} ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = -1$, $\texttt{factor} = -0.75\pi/e_k$).
- The magnetic energy per unit of volume e_k ($\vec{a}_1 = \vec{a}_2 = \vec{b}$, $j = 0$, $\texttt{factor} = 0.5$).
- The magnetic energy dissipation ϵ_b ($\vec{a}_1 = \vec{a}_2 = \vec{b}$, $j = 2$, $\texttt{factor} = \eta$).

2.7.2 CORREL_SCALAR

```
CALL correl_scalar(s1,s2,j,factor,result)
```

The input variables and input parameters are defined as

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy), INTENT(IN) ::  s1, s2
INTEGER, INTENT(IN) ::  j
REAL, INTENT(IN)    ::  factor
```

This subroutine computes the following quantity C (ℓ is assumed to be integer):

$$C = \frac{\texttt{factor}}{V} \int d\vec{x} \, \left(\Delta^{j/2-\ell} s_1(\vec{x})\right) \left(\Delta^{\ell} s_2(\vec{x})\right) , \tag{2.10}$$

$$= \frac{\texttt{factor}}{V} \int d\vec{x} \, \left(\Delta^{j/2-\ell} s_1(\vec{x})\right) \cdot \left(\Delta^{\ell} \partial_x^2 s_2(\vec{x})\right) , \tag{2.11}$$

and stores it into the real number `result1`. The equality between the above expressions is valid because TURBO is restricted to problems with periodic boundary conditions (see section 1.5 for details on how this type of integrals can be computed using TURBO). Examples of quantities that can be computed with this subroutine are:

- The correlation between two scalars ($j = 0$, `factor=1`).
- The fluctuation of a scalar quantity ($s_1 = s_2 = s$, $j = 0$, `factor=1`).
- The dissipation of fluctuations of a scalar quantity ($s_1 = s_2 = s$, $j = 2$, `factor=1`).

2.7.3 CORREL_HELICITY

```
CALL correl _helicity(s1,s2,j,factor,result)
```

The input variables and input parameters are defined as

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy,3), INTENT(IN) ::  a1, a2
INTEGER, INTENT(IN) ::  j
REAL, INTENT(IN)    ::  factor
```

This subroutine computes the following quantity C (ℓ is assumed to be integer):

$$C = \frac{\texttt{factor}}{V} \int d\vec{x} \, \left(\Delta^{j/2} \vec{a}_1(\vec{x}) \right) \cdot \left(\nabla \times \vec{a}_2(\vec{x}) \right) \, , \tag{2.12}$$

$$= \frac{\texttt{factor}}{V} \int d\vec{x} \, \Delta^{\ell} \vec{a}_1(\vec{x}) \cdot \left(\nabla \times \Delta^{j/2-\ell} \vec{a}_2(\vec{x}) \right) \, , \tag{2.13}$$

$$= \frac{\texttt{factor}}{V} \int d\vec{x} \, \vec{a}_1(\vec{x}) \cdot \left(\nabla \times \Delta^{j/2} \vec{a}_2(\vec{x}) \right) \, , \tag{2.14}$$

and stores it into the real number `result1`. The equality between the above expressions is valid because TURBO is restricted to problems with periodic boundary conditions (see section 1.5 for details on how this type of integrals can be computed using TURBO). Examples of quantities that can be computed with this subroutine are:

- The kinetic helicity per unit of volume h_k ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = 0$, `factor=1`).
- The magnetic helicity per unit of volume h_b ($\vec{a}_1 = \vec{a}_2 = \vec{b}$, $j = -2$, `factor=1`).
- The dissipation of kinetic helicity per unit of volume ϵ_{hk} ($\vec{a}_1 = \vec{a}_2 = \vec{u}$, $j = 2$, `factor`$= -2\,\nu$).
- The dissipation of magnetic helicity per unit of volume ϵ_{hb} ($\vec{a}_1 = \vec{a}_2 = \vec{b}$, $j = 0$, `factor`$= -2\,\eta$).

2.7.4 COMPUTE_DISS_QS

```
CALL compute_diss_QS(u,omegaQS)
```

The input variables are defined as

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy,3), INTENT(IN) ::  u
```

This subroutine computes the quantity `omegaQS` defined as:

$$\texttt{omegaQS} = \frac{1}{\vec{B}^2\, V} \int d\vec{x} \left(\frac{(\vec{B}.\nabla)^2}{\nabla^2} \right) \left(\vec{u}(\vec{x}) \cdot \vec{u}(\vec{x}) \right) ,$$

where $\vec{B}$ is a constant magnetic field. Its direction is not imposed and is prescribed by the values of the zero modes of the magnetic field that have to be given by the user in the namelist `mag_parameters`. This subroutine is useful to compute the Joule dissipation in the quasi-static approximation, after multiplication by the interaction parameter.

2.7.5 COMPUTE_ENERGY_REAL

```
CALL compute_energy_real(a,ener)
```

The input variables are defined as

```
COMPLEX, DIMENSION(0:cx,0:cz,0:cy,3), INTENT(IN) ::  a
```

This subroutine computes the quantity `ener` defined as:

$$\texttt{ener} = \frac{1}{2V} \int d\vec{x}\, \vec{a}(\vec{x}) \cdot \vec{a}(\vec{x})$$

The main purpose of this subroutine is to allow the user to verify the accuracy of the energy value computed using the **correl_vector** subroutine. As a consequence of the Parseval's theorem, the energy computed using the following two calls

```
CALL compute_energy_real(u,ener)
CALL correl_vector(u,u,0,0.5,ener)
```

should be the same. However, because it requires calls to FFT subroutines, the subroutine **compute_energy_real** is much more time consuming and the use of the subroutine **correl_vector** should definitively be preferred.

2.7.6 COMPUTE_RLAMBDA

```
CALL compute_Rlambda(energy,dissipation,Rlambda)
```

This very simple subroutine computes the quantity `Rlambda`, known as the Taylor-scale Reynolds number, defined as:

$$\texttt{Rlambda} = \frac{u'\lambda}{\nu}$$

where u' represents the root mean square of one component of the velocity field. In isotropic turbulence, the choice of the component, u_x, u_y or u_z should not influence the value of u'. However, in anisotropic turbulence, it is preferable to replace it by a function of the total energy $u' = \left(\frac{2E}{3}\right)^{1/2}$. The Taylor microscale is defined by $\lambda = \left(\frac{10\nu E}{\varepsilon}\right)^{1/2}$. The parameter `Rlambda` is thus given by

$$\texttt{Rlambda} = \left(\frac{5}{3\nu\varepsilon}\right)^{1/2} 2E$$

where ν is the viscosity, ε is the dissipation and E is the kinetic energy. It is important to note, however, that this parameter may give a very poor diagnostics in strongly anisotropic turbulence since it is build using quantities that mix information from all three components of the velocity field.

2.7.7 COMPUTE_KMAXETA

```
CALL compute_kmaxeta(dissipation,kmaxeta)
```

This simple subroutine is used to evaluate the accuracy of the run. In isotropic Navier-Stokes turbulence, the dimensionless product of the largest wave vector $k_{\max}$ and of the Kolmogorov dissipation length η, defined as

$$\eta = (\frac{\nu^3}{\varepsilon})^{1/4}$$

where ε is the dissipation, is supposed to be of the order (or larger than) 1.5. In this case, it is usually considered that a sufficiently large fraction of the dissipation range is numerically resolved. Of course, this is only true for relatively low order of the statistics of the velocity field and of its low order derivatives. The subroutine **compute_kmaxeta** simply computes this product. The output is the product $k_{\max}\ \eta$. It is important to note, however, that this parameter may give a very poor diagnostics on the accuracy of the simulation in strongly anisotropic turbulence since it is build using quantities that mix information from all three components of the velocity field.

SECTION 2.8

THE SUBROUTINES IN THE FILE: TT_UPDATE.F90

This file contains the subroutines used for computing the evolution of the variables and computing the time step. TURBO can use two different methods for removing the aliasing errors: the phase shift method and the two-third $(2/3)$ truncation method (see 1.7.2). The subroutine **update_variables** chooses which of this methods is going to be used in function of the `dealias` switch value. The file *tt_update.f90* contains the following subroutines:

- **update_variables**: Updates all the variables using the appropriate Runge-Kutta scheme.
- **rk3_twothird**: Runge-Kutta subroutine using the two-third dealiasing method.
- **rk4_shifts**: Runge-Kutta subroutine using the phase shift dealiasing method.
- **rk_step**: Runge-Kutta step.
- **rk_shiftgrid**: Computes the effect of a grid shift on all the variables.
- **rk_nonlin**: Computes the nonlinear terms in the Runge-Kutta scheme.
- **rk_nonlin_add**: Intermediate subroutine in the Runge-Kutta scheme.
- **rk_forcing**: Computes the forcing term.
- **rk_expon**: Computes the exponential factors due to linear (dissipative as well as rotation) terms.
- **rk_endstep**: Runge-Kutta end step for both dealiasing methods.
- **compute_dt**: computes the time step.

2.8.1 UPDATE_VARIABLES

```
CALL update_variables(iter)
```

This subroutine is called by the main TURBO program and is used to compute the variables at time $t+dt$ from the knowledge of the variables at time t. It requires work arrays `dvel`, `dmag` and `dscal` that are allocated only if they are needed depending on the value of the logicals `VELEQ`, `MAGEQ` and `SCAEQ`. It also initializes to zero `inj_energy` and `inj_helicity` if `VELEQ` is true. These two parameters are used to compute the energy and helicity injection when a forcing mechanism is used in the velocity equation.

The subroutine then calls another subroutine depending on the value of the input parameter `dealias`. If `dealias=1`, a four step Runge-Kutta algorithm based on a grid shifting is used and is implemented in the subroutine **`rk4_shift`**. If `dealias=2`, a three step Runge-Kutta algorithm based on the two-third truncation method is used and is implemented in the subroutine **`rk3_twothird`**. Both these techniques are described in the section 1.7.2 while the Runge Kutta algorithm is discussed in section 1.7.1.

At the end, the symmetry properties needed to insure that the Fourier modes do correspond to real scalar and vector fields are imposed by a call to the subroutine **`symmetrize_all`** 2.5.13.

2.8.2 RK3_TWOTHIRD

```
CALL rk3_twothird()
```

This subroutine is used to compute the updated variables using a three step Runge-Kutta algorithm based on the two-third truncation method (see 1.7.1). The parameters used in the Runge-Kutta scheme are defined in 1.68. They are declared at the beginning of the subroutine.

The Runge-Kutta algorithm is described by the equations 1.65 and 1.66 which we reproduce here for convenience:

$$\left\{ \begin{array}{ll} k_i & = e^{-\nu\xi_i h}\ (k_{i-1} + \gamma_i\ N(y_{i-1})) \\ y_i & = e^{-\nu\xi_i h}\ (y_{i-1} + \alpha_i\ h\ N(y_{i-1})) + \beta_i\ k_i\ h \end{array} \right. \tag{2.15}$$

Each of these steps can be decomposed into three manipulations. First, the nonlinear term $N(y)$ has to be computed and added to both k and y as follows:

$$\left\{ \begin{array}{ll} k & \rightarrow k + \gamma_i\ N(y) \\ y & \rightarrow y + \alpha_i\ h\ N(y) \end{array} \right. \tag{2.16}$$

This is achieved by calling the subroutine **rk_nlstep(al,ga,substep)** 2.8.4. This subroutine actually also includes the computation of external forcing terms if they are not linear functions of the velocity. This excludes the Lorentz force in the quasi-static approximation and the Coriolis force when a global rotation is imposed which are treated analytically through the exponential factors, as well as the linear dissipative terms (viscosity, resistivity, diffusion):

$$\left\{ \begin{array}{ll} k & \rightarrow e^{-\nu\xi_i h} k \\ y & \rightarrow e^{-\nu\xi_i h} y \end{array} \right. \tag{2.17}$$

This is achieved by calling the subroutine **rk_expon(fac)** 2.8.8. The real number `fac` represents the product $\xi_i\ h \equiv \xi_i\ dt$. Finally, the term $\beta_i\ k_i\ h$ has to be added:

$$y \rightarrow y + \beta_i\ k_i\ h \tag{2.18}$$

This is achieved into the subroutine **rk_endstep()** 2.8.9.

2.8.3 RK4_SHIFT

```
CALL rk4_shift(iter)
```

This subroutine is used to compute the updated variables using a four step Runge-Kutta algorithm based on the grid shifting method (see 1.7.1). The parameters used in the Runge-Kutta scheme are defined in 1.69. They are declared at the beginning of the subroutine.

As explained in Section 1.7.1, the total removal of the aliasing error would require 8 evaluations of the nonlinear terms on 8 different grids at each substep of the Runge-Kutta scheme. This would be prohibitively expensive. The strategy adopted here is to evaluate the nonlinear terms on the 8 different grids over two successive time steps, each of them having 4 substeps. The Runge-Kutta scheme is designed so that each nonlinear contribution from the 4 substeps enters in the evaluation of the updated variables with the same weight, at least at the lowest order in dt. Moreover, the time step dt is kept unchanged over two successive time steps so that 8 successive evaluations of the nonlinear terms contribute with the same weight to the updated variables, at lowest order in dt.

However, since these eight evaluations are not performed at exactly the same time, this procedure does not remove totally the aliasing error but increase its order by a factor dt. The aliasing error is further decreased by using random shifts modified every two time steps. In practice, these random shifts introduce a random noise that multiplies only the aliasing error and thus reduces its impact on the accuracy of the algorithm. An integer `odd_even` is defined to determine if the iteration number `iter` is odd (`odd_even=0`) or even (`odd_even=1`). Since the first iteration corresponds to `iter=1`, the random shifts have to be computed if and only if (`odd_even=0`). The random shifts have to be the same on all the nodes. They are thus computed only on the node with `myid=ioid` and then broadcasted to all the other nodes.

Since the Runge-Kutta scheme has been chosen with $\xi_2 = \xi_3 = 0$, there is no call to **`rk_expon`** in the second and third substeps of the algorithm.

2.8.4 RK_NLSTEP

```
CALL rk_nlstep(al,ga,substep)
```

This subroutine computes the nonlinear terms and the forcing (except the Lorentz force in the quasi-static approximation and the Coriolis force when a global rotation is imposed which are taken into account in the exponential factors in the subroutine **rk_expon**).

The first part of the subroutine is used only when dealiasing is performed using grid shifting. It is used to define the shifts used in the different substeps. They correspond to random shifts `rsx`, `rsy` and `rsz` defined in the subroutine **rk4_shift** and systematic shifts corresponding to a quarter of grid spacing Δ in each direction (so that each grid is shifted by half a grid spacing in at least one direction when compared to another grid). These systematic shifts are defined as follows:

$$\begin{array}{lll}
\texttt{sx} = +\Delta_x/4, \texttt{sy} = +\Delta_y/4, \texttt{sz} = +\Delta_z/4 & \text{odd value of } \texttt{iter} & \texttt{substep=1} \\
\texttt{sx} = -\Delta_x/4, \texttt{sy} = +\Delta_y/4, \texttt{sz} = +\Delta_z/4 & \text{odd value of } \texttt{iter} & \texttt{substep=2} \\
\texttt{sx} = +\Delta_x/4, \texttt{sy} = -\Delta_y/4, \texttt{sz} = +\Delta_z/4 & \text{odd value of } \texttt{iter} & \texttt{substep=3} \\
\texttt{sx} = +\Delta_x/4, \texttt{sy} = +\Delta_y/4, \texttt{sz} = -\Delta_z/4 & \text{odd value of } \texttt{iter} & \texttt{substep=4} \\
\texttt{sx} = -\Delta_x/4, \texttt{sy} = -\Delta_y/4, \texttt{sz} = -\Delta_z/4 & \text{even value of } \texttt{iter} & \texttt{substep=1} \\
\texttt{sx} = +\Delta_x/4, \texttt{sy} = -\Delta_y/4, \texttt{sz} = -\Delta_z/4 & \text{even value of } \texttt{iter} & \texttt{substep=2} \\
\texttt{sx} = -\Delta_x/4, \texttt{sy} = +\Delta_y/4, \texttt{sz} = -\Delta_z/4 & \text{even value of } \texttt{iter} & \texttt{substep=3} \\
\texttt{sx} = -\Delta_x/4, \texttt{sy} = -\Delta_y/4, \texttt{sz} = +\Delta_z/4 & \text{even value of } \texttt{iter} & \texttt{substep=4}
\end{array} \tag{2.19}$$

The random and systematic shifts are then summed into the integers `sx`, `sy` and `sz` and applied by a call to the subroutine **rk_shiftgrid** 2.8.5.

```
CALL rk_shiftgrid(sx,sy,sz)
```

The subroutine then computes the nonlinear terms $\partial_j(u_i u_j)$, $\partial_j(u_i b_j)$, $\partial_j(b_i b_j)$ or $\partial_j(c_\alpha u_j)$ depending on the nature of the problem. These nonlinear terms are computed and added to the variables and to the work arrays respectively with the weights `al dt` and `ga` by the call

```
CALL rk_nonlin(al,ga)
```

The lines of code that follow this call are used to remove the aliasing error. If the grid shifting procedure is used, the variables are re-expressed on the original grid. If the two-third truncation method is used, the Fourier modes that have to be removed are set to zero by multiplying the variables and the work arrays by an pre-defined array `zeros_twothird`, initialized in **init_wavevectors** 2.5.3. It is either 1 if the wave vector has to be retained or 0 if it has to be set to zero.

Next, the flow incompressibility condition ($\nabla \cdot \vec{u} = 0$) and the magnetic field zero divergence ($\nabla \cdot \vec{b} = 0$) are enforced for the main (`vel`, `mag`) and work arrays (`dvel`, `dmag`).

The last part is used to impose the forcing that would not be accounted for in the exponential factor. Two distinct ways of forcing the fluid are implemented. In the first method, an extra term is added to the right-hand side of the Navier-Stokes equation (cases `FORCEPARA=1,3,4`). It acts as an additional mechanical force. In the last substep of the Runge-Kutta method, the energy and helicity injection rates are also computed. They are given respectively by the correlation of the force with the velocity and the vorticity. The second method consists in modifying directly the velocity field, usually by keeping constant some of its properties, such as the velocity profile or the energy in a range of wave vectors (cases `FORCEPARA=2,5`). In that case, the energy and helicity injection rates are

computed directly by measuring these quantities before and after the modifications imposed to the velocity. These differences are accumulated over all the substeps.

2.8.5 RK_SHIFTGRID

```
CALL rk_shiftgrid(shiftx,shifty,shiftz)
```

This subroutine is very simple. It computes the factors that multiply the Fourier modes when the computational grid is shifted and multiplies all the variables by these factors, depending of course of the value of the logicals VELEQ, MAGEQ and SCAEQ. The input real numbers correspond to the shifts:

```
REAL , INTENT(IN) ::  shiftx, shifty, shiftz
```

The factors are computed in the array expo and correspond to

$$e^{i(k_x \texttt{shiftx} + k_y \texttt{shifty} + k_z \texttt{shiftz})} \tag{2.20}$$

2.8.6 RK_NONLIN

```
CALL rk_nonlin(al,ga)
```

This subroutine is the core of the TURBO code. Indeed, it computes the nonlinear term and adds it to the variables and to the work arrays with weights `al*dt` and `ga` respectively as described by the equation 2.16.

Firstly, the subroutine allocates the arrays `ur1, ur2, ur3` and `br1, br2, br3` for the expression of the velocity and the magnetic field in real space if required (depending on the values of `VELEQ` and `MAGEQ`) and makes the inverse Fourier transforms. It then computes, again if needed by the type of simulation, the nonlinear terms and adds them to the variable arrays `vel`, `mag` and `scal` and to the work arrays `dvel`, `dmag` and `dscal`. The nonlinear terms have always the structure $\partial_k(q_{l_1}q_{l_2})$. For instance, the product $u_x u_y$ appears in the right-hand-side of the evolution equations for both u_x ($\partial_y u_x u_y$) and u_y ($\partial_x u_x u_y$). This is combined in the following lines:

```
prod = ur1*ur2
IF (MAGEQ) prod=prod-br1*br2
CALL fft3ds(prod,c1)
CALL rk_nonlin_add(c1,vel(:,:,:,1),dvel(:,:,:,1),al*dt,ga,2)
CALL rk_nonlin_add(c1,vel(:,:,:,2),dvel(:,:,:,2),al*dt,ga,1)
```

The first line simply computes the product of $u_x u_y$ and stores it into the work array `prod`. The second line subtracts the product $b_x b_y$ if `MAGEQ` is true. The third line transforms the product in Fourier space. The fourth line uses the subroutine **`rk_nonlin_add`** to add the derivative of this product with respect to y (this explains the last entry "2" of the subroutine call) to both `vel(:,:,:,1)` and `dvel(:,:,:,1)` with the weights `al*dt` and `ga` respectively. The last line is similar but uses a derivative with respect to x and updates `vel(:,:,:,2)` and `dvel(:,:,:,2)`.

The only trick used in this subroutine is based on the incompressibility of the velocity field. Since, both the velocity and the associated work array are imposed to be divergence free at the end of each substep, adding a gradient to the nonlinear term does not modify the evolution:

$$\partial_j u_i u_j - \partial_i \phi \Leftrightarrow \partial_j u_i u_j \tag{2.21}$$

The choice that is made in TURBO is $\phi = u_y u_y$. Hence, the term $u_x u_x$ and $u_z u_z$ are replaced by $u_x u_x - u_y u_y$ and $u_z u_z - u_y u_y$ while the term $u_y u_y$ has not to be computed. This trick saves the evaluation of one FFT. When `MAGEQ` is true, $\phi = u_y u_y - b_y b_y$.

2.8.7 RK_NONLIN_ADD

```
CALL rk_nonlin_add(c1,var,dvar,fac1,fac2,igrad)
```

This subroutine derives a nonlinear term that has already been Fourier transformed `c1` with respect to the space direction defined by `igrad`. It then adds the results to the variables `var` and to the work arrays `dvar` with weights `fac1=al*dt` and `fac2=ga` respectively as described by the equation 2.16. In practice, it does the following manipulations:

$$\left.\begin{array}{ll} \texttt{var} & \longrightarrow \texttt{var} + \texttt{fac1}\ \partial_x \texttt{c1} \\ \texttt{dvar} & \longrightarrow \texttt{dvar} + \texttt{fac2}\ \partial_x \texttt{c1} \end{array}\right\} \text{if } \texttt{igrad=1}$$
$$\left.\begin{array}{ll} \texttt{var} & \longrightarrow \texttt{var} + \texttt{fac1}\ \partial_y \texttt{c1} \\ \texttt{dvar} & \longrightarrow \texttt{dvar} + \texttt{fac2}\ \partial_y \texttt{c1} \end{array}\right\} \text{if } \texttt{igrad=2} \qquad (2.22)$$
$$\left.\begin{array}{ll} \texttt{var} & \longrightarrow \texttt{var} + \texttt{fac1}\ \partial_z \texttt{c1} \\ \texttt{dvar} & \longrightarrow \texttt{dvar} + \texttt{fac2}\ \partial_z \texttt{c1} \end{array}\right\} \text{if } \texttt{igrad=3}$$

Since `c1` is expressed in Fourier space, the derivatives with respect to x, y or z are obtained by multiplying by the arrays `ikx`, `iky` or `ikz`.

2.8.8 RK_EXPON

```
CALL rk_expon(fac)
```

This subroutine is needed because the Runge-Kutta algorithms that are implemented in TURBO are designed to produce the exact solutions in the absence of nonlinear terms. This is achieved by a change of variables 1.61 in which the transport coefficients ν, η are κ_s enter exponential factors. For instance, in each Runge-Kutta substep, both the magnetic field `mag` and the corresponding work array `dmag` are multiplied by a factor $\exp(-\eta k^2 \texttt{fac} dt)$.

In the quasi-static approximation, the Lorentz force can be expressed in Fourier space as a linear function of the velocity:

$$F_i^{\text{Lorentz}} = -\sigma \frac{(\mathbf{B}^{ext} \cdot \mathbf{k})^2}{\rho k^2} u_i \tag{2.23}$$

where ρ is the density, σ is the electric conductivity of the fluid and $\mathbf{B}^{ext}$ is an external magnetic field. The exponential factor are then modified and the function $-\nu k^2$ is replaced by $-\nu k^2 - (\mathbf{b}_0 \cdot \mathbf{k})^2/k^2$. The vector $\mathbf{b}_0 = \sigma \mathbf{B}^{ext}/\rho$ is provided through the namelist `mag_parameters`.

Finally, in the presence of a solid body rotation, the components of the velocity field and of the related work array have to be multiplied by the matrix $\tilde{D}(\omega\tau)$ before being multiplied by the exponential factors.

This subroutine is called in each Runge-Kutta substep in order to multiply the variables and work arrays by the proper exponential factors and by the matrix $\tilde{D}(\omega\tau)$ if needed.

2.8.9 RK_ENDSTEP

```
CALL rk_endstep(be)
```

The first purpose of this subroutine is to add the term proportional to β_i that appear in the relation 1.66 in each Runge-Kutta substep. However, because this addition has to be made at the end of the substep, the subroutines is also used to take care of various constraints on the variables:

- It impose the $\mathbf{k} = 0$ modes for the velocity field if `veleq` is true and for the magnetic field if `mageq` is true.
- As a consequence of the discussion in 1.4, the subroutine also imposes that the modes corresponding to `nx/2`, `nyx/2` or `nz/2` are set to zero.

2.8.10 COMPUTE_DT

```
CALL compute_dt()
```

This subroutine is used to compute the time step. If `fixed_dt=1`, then the time step is set to the prescribed value `dt_dat`. Both the parameters `fixed_dt` and `dt_dat` are provided to TURBO through the namelist `numerics_parameters`.

If `fixed_dt=0`, the time step is computed automatically and is given by the following formula (assuming that `VELEQ` is true):

$$\texttt{dt} = \frac{\texttt{cfl}}{\pi \left(\max(|u_x|)/dx + \max(|u_y|)/dy + \max(|u_z|)/dz \right)} \tag{2.24}$$

The coefficient `cfl` is also provided to TURBO through the namelist `numerics_parameters`. This is a special form of the Courant-Friedrichs-Lewy condition (CFL condition) that requests that the timestep must be less than the time for the fluid to travel adjacent grid points. If `MAGEQ` is true, the time step is the lowest value of this expression and a similar expression using the magnetic field.

SECTION 2.9

THE SUBROUTINES AND FUNCTIONS IN FILES: XX_EXTER.F90 & XX_MOD.F90

These file contain the subroutines, functions and modules that can be modified by the user.

- `external_forcing`: Computes the external force. It is called by the subroutine **`forcing`** when `FORCEPARA=3`. Default is `force=0.0`
- `external_modif_vel`: Imposes a number of external constraints on the velocity field. It is called by the subroutine **`modif_vel`** when `FORCEPARA=5`. By default no constraint is imposed.
- `external_init_vel`: Computes the initial velocity field as defined by the user. It is called by the subroutine **`init_vel_field`** when `init_vel=extern=3`. Default is `vel=0.0`.
- `external_init_mag`: Computes the initial magnetic field defined by the user. It is called by the subroutine **`init_mag_field`** when `init_mag=extern=3`. Default is `mag=0.0`.
- `external_init_scal`: Computes an initial scalar field defined by the user. It is called by the subroutine **`init_sca_field`** when `init_sca=2`. Default is `s=0.01*iscal`. It is called independently for each scalar field and the subroutines has two arguments: the scalar field itself and its index `iscal`.
- `external_energy_spectrum`: It is a function of the wavevector `k` and of four parameters `pa`, `pb`, `pc` and `pd`. The default implementation is the following function :

$$E(k) = \frac{\texttt{pa}\ k^4}{(\texttt{Ck}\ \texttt{Ns})^2}\ \exp\left(-2.0\ \left(\frac{k}{\texttt{pb}}\right)^2\right) \tag{2.25}$$

- `external_stats_init`: Initiates the various files needed for computing the statistics.
- `external_stats_end`: Deallocates the memory used for the statistics.
- `external_stats_write`: Save the statistics on files.
- `external_stats_compute`: Computes the statistics and displays on screen a series of global quantities. The default implementation only computes the kinetic (if `VELEQ=.true.`) energy and the magnetic energy (if `MAGEQ=.true.`).
- `module statmod`: A module that is foreseen for declaring variable used in the various subroutines dealing with the computation of statistics. It is declared in the file *`xx_mod.F90`*.

CHAPTER 3

RUNNING TURBO

To be written in a future release of this documentation file.

CHAPTER 4

POST PROCESSING TOOLS

To be written in a future release of this documentation file.

CHAPTER 5

TEST-CASE SIMULATIONS

We present a series of test performed by the TURBO team. These test are included in the manual to allow users to test the code on their machines and to test the new libraries implementations (notably any new FFT releases). Also, these tests build confidence in the code capability to correctly solve fully nonlinear turbulence.

SECTION 5.1

RK3 TEST FOR KOLMOGOROV FLOW

The TURBO solver makes use of a third order Runge-Kutta method (RK3) for the time advancement. We test the accuracy of the current implementation by comparing the numerical results obtained with the exact solution for a analytically solvable problem. One such test problem is represented by a nonconductive Kolmogorov flow in the laminar regime. Although numerically we solve the nonlinear term, the laminar regime allows us to take as starting point for our analytical study the linearized incompressible Navier-Stokes equation:

$$\partial_t u_i = \nu \nabla^2 u_i + F_i \tag{5.1}$$

where we have suppressed the space and time dependency notations and the index i stands for the x, y and z components of the fields. The Kolmogorov flow is generated by a stationary forcing which varies sinusoidally in space (F_i). We assume the forcing to be aligned with the x axis and modulated along the y axis. We denote the amplitude of the force by A.

$$F_x = A\sin(k_f y) \tag{5.2}$$
$$F_y = F_z = 0 \tag{5.3}$$

Since the TURBO code is a spectral solver we rewrite equation 5.1 in wave-number space:

$$\partial_t \tilde{u}_i = -\nu k^2 \tilde{u}_i + \tilde{F}_i \tag{5.4}$$

In the laminar regime the forcing generates a unidirectional flow parallel to x axis. From equation 5.4 we see the asymptotic solution ($\partial_t \tilde{u}_i \approx 0$) for the laminar Kolmogorov flow is:

$$\tilde{u}_x^{asymptotic} = \frac{\tilde{F}_x}{\nu k^2} \tag{5.5}$$

At this step it is useful to determine the Kolmogorov force in wave-number space using TURBO discrete space representation introduced in *CHAPTER 1*. In this notation 5.2 becomes:

$$F_x^d(\vec{i}) = A\sin(\vec{p}_f \cdot \vec{i}) \tag{5.6}$$

where $\vec{p}_f = (0, k_f, 0)$. The discrete forward Fourier transform (for $s = 1$) gives us:

$$\begin{aligned}
\tilde{F}_x^d(\vec{p}) &= C_k \sum_{\vec{i}} F^d(\vec{i}) \exp[-i\vec{p}\cdot\vec{i}] \\
&= C_k \sum_{\vec{i}} A\sin(\vec{p}_f\cdot\vec{i}) \exp[-i\vec{p}\cdot\vec{i}] \\
&= C_k \sum_{\vec{i}} \frac{A}{2i}\left(\exp[i\vec{p}_f\cdot\vec{i}] - \exp[-i\vec{p}_f\cdot\vec{i}]\right)\exp[-i\vec{p}\cdot\vec{i}] \\
&= -\frac{iA}{2} C_k \sum_{\vec{i}} \left(\exp[i(\vec{p}_f - \vec{p})\cdot\vec{i}] - \exp[i(-\vec{p}_f - \vec{p})\cdot\vec{i}]\right) \\
&= -\frac{iA}{2} C_k \sum_{\vec{i}} \left(N\delta_{\vec{p},\vec{p}_f} - N\delta_{\vec{p},-\vec{p}_f}\right)
\end{aligned} \tag{5.7}$$

where $\delta_{\vec{p},\vec{p}_f} = 1$ for $\vec{p} = \vec{p}_f$.and zero otherwise. Since $\tilde{F}_x^d(\vec{p})^* = \tilde{F}_x^d(-\vec{p})$ we have $\tilde{F}_x^d(\vec{p}_f) = -\frac{iA}{2}NC_k$ and zero for the other values of $\vec{p}$. From the asymptotic solution given by 5.5 and the Kolmogorov force just determined we find:

$$\tilde{u}_x^{asymptotic}(0, k_f, 0) = \frac{\tilde{F}_x(0, k_f, 0)}{\nu k_f^2} = -\frac{iA}{2\nu k_f^2}NC_k \tag{5.8}$$

For a given set of parameters we will look at the difference between the analytical and the numerical asymptotic solution ($\tilde{u}_x^{asymptotic}(0, k_f, 0)$) for a series of fixed time steps (dt). For the RK3 method the errors per time step are of order dt^4 and the total error is of order dt^3. This implies that the difference between the numerical and analytical asymptotic solutions should behave as the dt^3. To better see the difference we will compare the quantity U defined as:

$$U \equiv Im\left[\frac{\tilde{u}_x^{asymptotic}(0, k_f, 0)}{NC_k}\right] = -\frac{A}{2\nu k_f^2} \tag{5.9}$$

For the current FFT implementation TURBO sets $C_k = 1$. Since we are interested in a laminar solution we use a small number of modes and high viscosity levels. The simulation box is $(2\pi)^3$.and the set of parameters chosen are:

N	k_f	A	ν
16^3	1	1	1

From the above parameters we find $U_{analytic} = -0.5$. Using DOUBLE PRECISION simulations we start from the same initial conditions and perform three runs for three different fixed time steps. The U values found for the three runs and the difference from the analytical value are:

n	dt	U	$U - U_{analytic}$	$\frac{F}{216}\left(\nu k^2\right)^3 dt^3$
1.	10^{-1}	-0.499997693542029	$2.30645797116846 \cdot 10^{-6}$	$2.31481481481482 \cdot 10^{-6}$
2.	10^{-2}	-0.499999997685974	$2.31402585981755 \cdot 10^{-9}$	$2.31481481481481 \cdot 10^{-9}$
3.	10^{-3}	-0.499999999997732	$2.26807461700673 \cdot 10^{-12}$	$2.31481481481481 \cdot 10^{-12}$

The last column represent the error formula found by a Maple implementation of the RK3 scheme for the linear equation compared to the analytical solution. In the table we present it as being proportional to dt^3 instead of dt^4 to take into account the error accumulation in time. There is a good agreement between the values found by the Maple program and the values obtained by our code. We also see that by decreasing the time step by an order of 10 we decrease the error by an order of 1000. This test proves the correct implementation of the RK3 time advancement method.