Fourier Modes of a Real Scalar Field

Jorge L. deLyra
Department of Mathematical Physics
Physics Institute
University of São Paulo

Version 1, June 2006

We will consider here the technical questions of how to deal with the Fourier modes of a single-component real scalar field in numerical simulations. This includes how to organize them in order to avoid over-counting the degrees of freedom, how to index them into vectors and how to build the transformations from position space to momentum space and back in an efficient way, so that they can be executed very fast, for use in stochastic simulations. It is important to note that the type of implementation presented here is *not* what is usually known as the Fast Fourier Transform (FFT). It may be described rather as an Indexed Fourier Transform, and is meant specifically for the relatively small lattices used in the stochastic simulations of field-theoretical models in space-time dimensions ranging from $d = 2$ to $d = 5$.

1 Organizing the Degrees of Freedom

Just as we can represent the field in position space as a N^d-dimensional real vector, so we would like to do the same in momentum space. A real field φ on a N-lattice in d dimensions corresponds to N^d degrees of freedom, that is, to N^d independent real variables $\varphi(\vec{n})$, one for each site of the lattice, which is described by the integer coordinates $\vec{n}$. Since Fourier transformation is simply a change of basis in the space of functions on the lattice, from the position space representation $\varphi(\vec{n})$ to the momentum-space representation $\widetilde{\varphi}(\vec{k})$, it follows that the Fourier components must also include exactly N^d independent variables, one for each mode of the lattice, which is described by the integer coordinates $\vec{k}$. However, in its usual representation the Fourier transform of a real scalar field is complex,

$$\widetilde{\varphi}(\vec{k}) = \frac{1}{N^d} \sum_{\vec{n}} e^{\imath \frac{2\pi}{N} \vec{k} \cdot \vec{n}} \varphi(\vec{n}),$$

and therefore seems to include twice as many real variables. This is not really so because not all such variables are independent, due to the identity

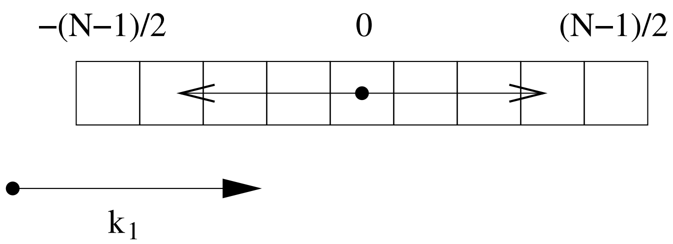
$$\widetilde{\varphi}(-\vec{k}) = \widetilde{\varphi}^*(\vec{k}),$$

or, using the decomposition $\widetilde{\varphi}(\vec{k}) = \mathfrak{R}(\vec{k}) + \imath\mathfrak{I}(\vec{k})$, the identities

$$\mathfrak{R}(-\vec{k}) = \mathfrak{R}(\vec{k}) \text{ and } \mathfrak{I}(-\vec{k}) = -\mathfrak{I}(\vec{k}).$$

Besides, there are some modes for which the transform is real, such as $\widetilde{\varphi}(\vec{0})$, for which $\mathfrak{I}(\vec{0}) = 0$. So we see that counting and keeping the truly independent variables is not so straightforward a task.

In order to keep just the N^d independent variables, we must first identify which modes are real and which modes are complex, and then adopt some standard form of keeping the real and imaginary parts in a unique way. A mode is real when $\widetilde{\varphi}(-\vec{k}) = \widetilde{\varphi}(\vec{k})$ identically, that is, for any field $\varphi(\vec{n})$, and is complex otherwise. For example, it is clear that $\widetilde{\varphi}(\vec{0})$ is real, as one can easily verify by inspection of the definition of the transform. Since it is not true in general that $\varphi(-\vec{n}) = -\varphi(\vec{n})$, one can also see from the definition that there are no purely imaginary modes. Hence we have two classes of modes, purely real modes that contain a single independent real variable each, and complex modes that contain two mutually independent real variables each. These complex modes are related in pairs by the identity above, so if we keep all of them we will be double-counting the variables associated to them. For odd N these pairings can be illustrated by the diagram below, representing the conjugate lattice in dimension $d = 1$,



where every square represents a mode and one can see a vector $\vec{k}$ as well as the negative of the same vector. In two dimensions we have the corresponding diagram in figure 1. In these diagrams one can see the limits of the range of the momentum coordinates for odd N, which are $-(N-1)/2$ and $(N-1)/2$. The zero mode obviously maps to itself. We see now that for odd N there is a single real mode, the zero mode, because all other modes map to a different mode inside the conjugate lattice, by the inversion of the vector $\vec{k}$, due to the fact that the extremes of the range are symmetrical. For even N the situation is more complicated, in the case $d = 1$ we have

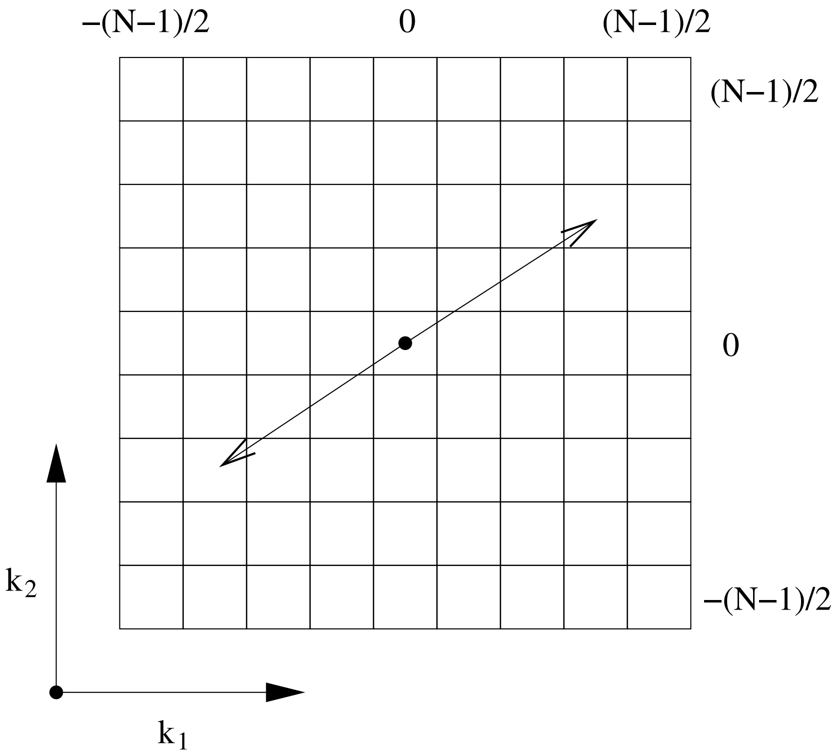
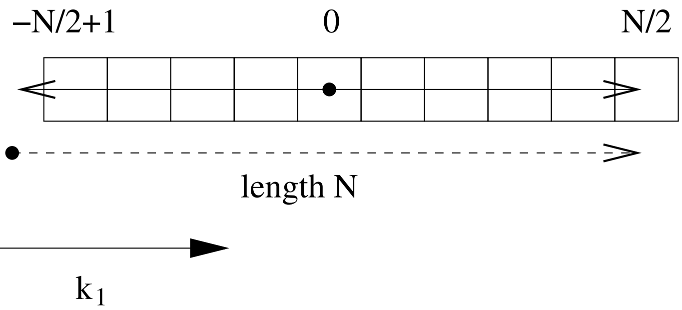


Figure 1: Inversion of a vector $\vec{k}$ in the conjugate lattice for odd N and $d = 2$.



So we see that, because the extremes of the range of the variable are no longer symmetrical, there is a mode, the mode $k_1 = N/2$, that maps to a point outside the conjugate lattice, so that we must then add to the inverted vector a period vector of length N, as shown in the figure, in order to bring it back into the conjugate lattice. As a consequence of all this the mode $k_1 = N/2$ also maps to itself, and is therefore a real mode, as one may check by direct inspection of the definition of the Fourier transform. In the case $d = 2$ we have the diagram in figure 2, were we see that in general many modes will map to points outside the conjugate lattice, namely all those modes that have at least one component equal to $N/2$, and in such cases we must add a period vector of length N along one of the directions of the lattice, as shown in the example depicted, with the period vector $(N, 0)$. In some cases it may be necessary to add several period vectors in different directions of the lattice, as in the case of the mode $(N/2, N/2)$ of this example, which needs two period vectors,

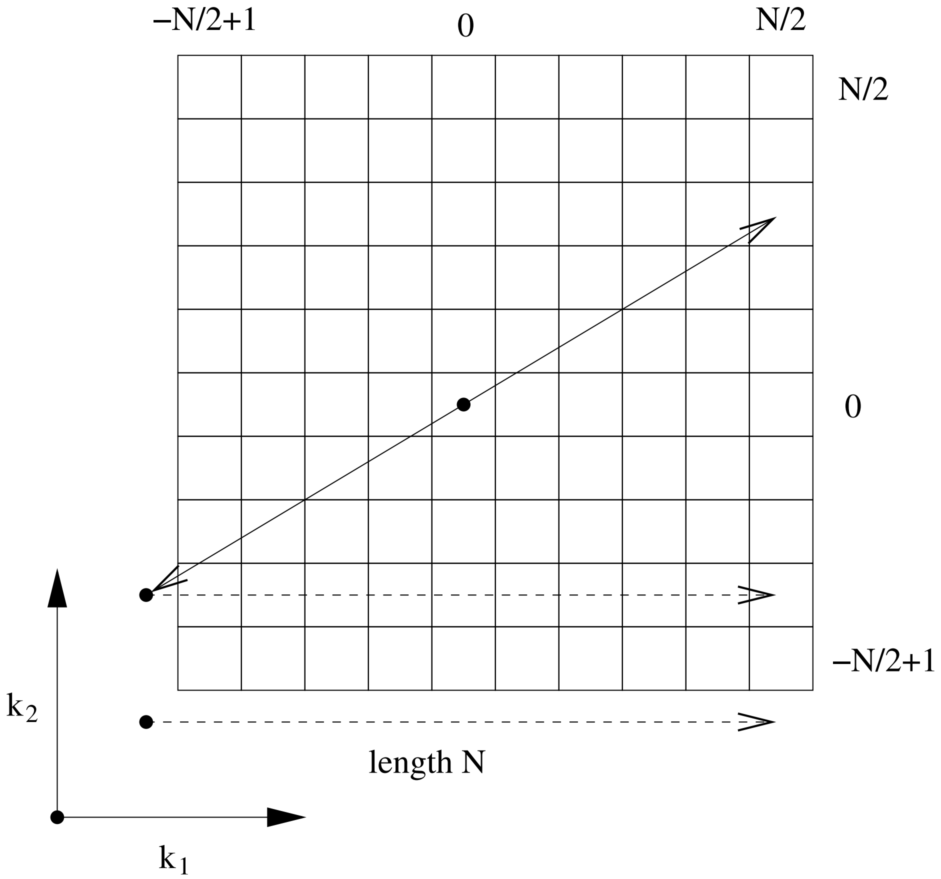
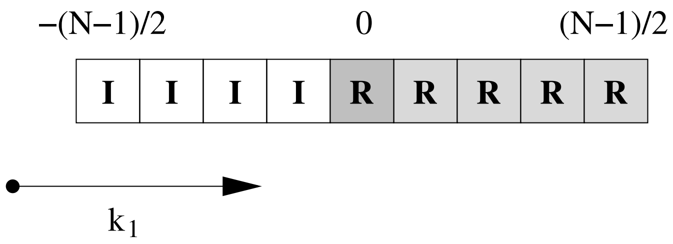


Figure 2: Inversion of a vector $\vec{k}$ in the conjugate lattice for even N and $d = 2$.

$(0, N)$ and $(N, 0)$. However, not all such modes are real, because most times they map to a different mode on the border of the lattice, as shown in the example. In the case $d = 2$ we can see from the diagram that there will be four real modes: $(0, 0)$, $(0, N/2)$, $(N/2, 0)$ and $(N/2, N/2)$.

In general we have 2^d real modes for even N, and just one for odd N. For odd N each complex mode is paired with the mode obtained by the inversion of $\vec{k}$, but for even N the pairings are not always so simple, as there are modes paired within the surfaces with one component equal to $N/2$. Nevertheless, all complex modes are dully paired in this way. The idea for the classification of the N^d independent variables is that we leave the purely real modes alone and, for each pair of conjugated complex modes, we keep only one real part and one imaginary part. In order to have a conjugate lattice with one real variable per site, just as have in position space for $\varphi(\vec{n})$, we must classify the modes in conjugate pairs and keep only the real part of one member of the pair, and only the imaginary part of the other member of the pair. As we do this we must also keep in mind that later on we will index all the modes into a single N^d-dimensional vector, just as we do in position space.

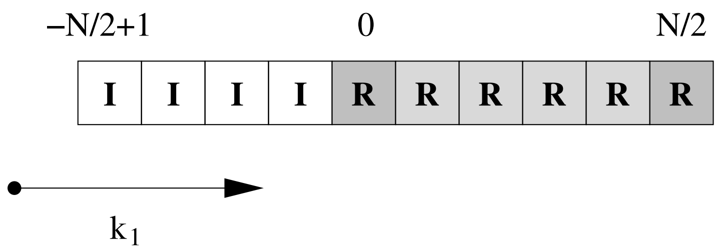
On order to describe the scheme we adopt for this classification of independent variables, we must discuss one dimension at a time. Starting with $d = 1$, in this case we may simply keep the real parts of the modes with positive components, and the imaginary parts of the modes with negative components. In the simpler odd-N case we represent this by the diagram



where the purely real modes are presented in a heavier shade, the complex modes of which we keep only the real parts in a lighter shade, and the complex modes of which we keep only the imaginary parts in white. The algorithm for implementing this is very simple,

$$
\begin{array}{ccl}
-(N-1)/2 \leq k_1 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \leq k_1 \leq (N-1)/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}
$$

In the even-N case, still for $d = 1$, we have the diagram



and the algorithm is only slightly different,

$$
\begin{array}{ccl}
-N/2+1 \leq k_1 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \leq k_1 \leq N/2-1 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_1 = N/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}
$$

For $d = 2$, in the odd-N case, we adopt a scheme which is a generalization of the previous one, and which is given by the diagram in figure 3. We start by scanning the lattice vertically by row, and define the situation on all horizontal rows that do not contain a real mode. For each row that contains a real mode we must examine the next coordinate. In this way we use the previous $d = 1$ scheme for the middle horizontal row of modes. The algorithm in this case is a two-fold iteration of the algorithm for $d = 1$,

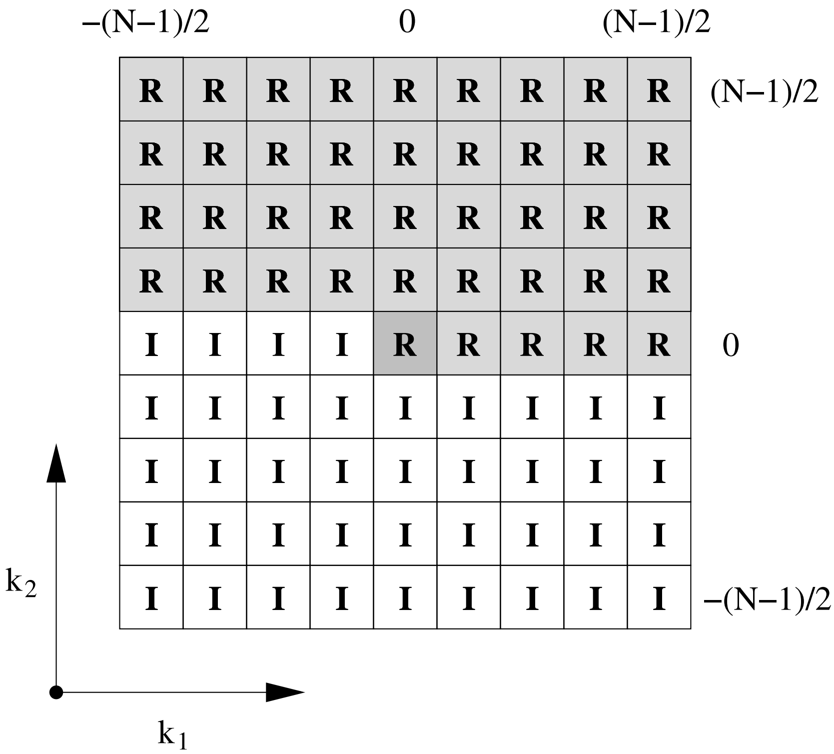


Figure 3: Classification of the momentum variables for odd N and $d = 2$.

$$
\begin{array}{ccl}
-(N-1)/2 \leq k_2 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_2 = 0 & \longrightarrow & \text{go to block for } k_1, \\
1 \leq k_2 \leq (N-1)/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}),
\end{array}
$$

$$
\begin{array}{ccl}
-(N-1)/2 \leq k_1 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \leq k_1 \leq (N-1)/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}
$$

Note that we start by checking k_2, not k_1, and only if it is zero we go on to check k_1. In the even-N case, still for $d = 2$, we adopt a similar scheme, given by the diagram in figure 4. Note that in this case the $d = 1$ scheme is used for two horizontal rows, the middle one and the top one. The algorithm in this case is once more a two-fold iteration of the algorithm for $d = 1$,

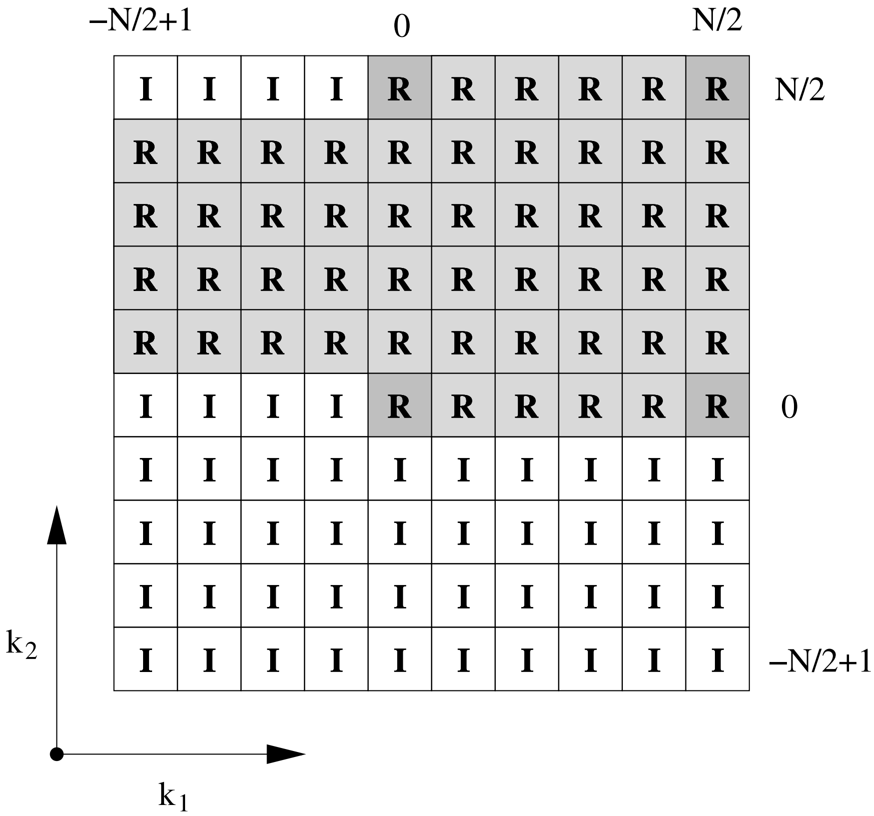


Figure 4: Classification of the momentum variables for even N and $d = 2$.

$$
\begin{array}{ccl}
-N/2+1 \leq k_2 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_2 = 0 & \longrightarrow & \text{go to block for } k_1, \\
1 \leq k_2 \leq N/2-1 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_2 = N/2 & \longrightarrow & \text{go to block for } k_1, \\
 & & \\
-N/2+1 \leq k_1 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \leq k_1 \leq N/2-1 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_1 = N/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}
$$

Note that in this case both the value $k_2 = 0$ and the value $k_2 = N/2$ send us to test the next coordinate, so that there are two paths to get from one block to the next block. Both the odd-N and the even-N algorithms can be easily generalized to arbitrary dimensions. Here is the general algorithm for odd N,

$$\begin{array}{ccl}
-(N-1)/2 \leq k_d \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_d = 0 & \longrightarrow & \text{go to block for } k_{d-1}, \\
1 \leq k_d \leq (N-1)/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
 & & \\
-(N-1)/2 \leq k_{d-1} \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_{d-1} = 0 & \longrightarrow & \text{go to block for } k_{d-2}, \\
1 \leq k_{d-1} \leq (N-1)/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
 & & \\
\ldots & \ldots & \ldots \\
 & & \\
-(N-1)/2 \leq k_1 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \leq k_1 \leq (N-1)/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}$$

Similarly, for even N we have the general algorithm

$$\begin{array}{ccl}
-N/2+1 \leq k_d \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_d = 0 & \longrightarrow & \text{go to block for } k_{d-1}, \\
1 \leq k_d \leq N/2-1 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_d = N/2 & \longrightarrow & \text{go to block for } k_{d-1}, \\
 & & \\
-N/2+1 \leq k_{d-1} \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_{d-1} = 0 & \longrightarrow & \text{go to block for } k_{d-2}, \\
1 \leq k_{d-1} \leq N/2-1 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_{d-1} = N/2 & \longrightarrow & \text{go to block for } k_{d-2}, \\
 & & \\
\ldots & \ldots & \ldots \\
 & & \\
-N/2+1 \leq k_1 \leq 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \leq k_1 \leq N/2-1 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_1 = N/2 & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}$$

Note that, except for the changes in the minimum and maximum values of each momentum coordinate, this algorithm includes the previous one, since for odd N a momentum coordinate will never have the value $N/2$. Hence, if we define k_l to be $k_l = (N-1)/2$ for odd N and $k_l = N/2 - 1$ for even N, and define also k_L to be $k_L = (N+1)/\!\!/2$ where the division indicated by the symbol $/\!\!/$ is integer division, with truncation, so that k_L is in fact $(N+1)/2$ for odd N and $N/2$ for even N, then we have the general algorithm, valid for both odd and even N,

$$\begin{array}{ccl}
-k_l \le k_d \le 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_d = 0 & \longrightarrow & \text{go to block for } k_{d-1}, \\
1 \le k_d \le k_l & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_d = k_L & \longrightarrow & \text{go to block for } k_{d-1}, \\
 & & \\
-k_l \le k_{d-1} \le 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_{d-1} = 0 & \longrightarrow & \text{go to block for } k_{d-2}, \\
1 \le k_{d-1} \le k_l & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_{d-1} = k_L & \longrightarrow & \text{go to block for } k_{d-2}, \\
 & & \\
\ldots & \ldots & \ldots \\
 & & \\
-k_l \le k_1 \le 1 & \longrightarrow & \text{calculate and store } \Im(\vec{k}), \\
k_1 = 0 & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
1 \le k_1 \le k_l & \longrightarrow & \text{calculate and store } \Re(\vec{k}), \\
k_1 = k_L & \longrightarrow & \text{calculate and store } \Re(\vec{k}).
\end{array}$$

In the way of verification, one can count the number of modes which are classified in each case. For odd N we have, at each step of the algorithm,

$$\begin{array}{ccl}
-(N-1)/2 \le k_d \le 1 & \longrightarrow & N^{d-1}(N-1)/2, \\
k_d = 0 & \longrightarrow & \text{proceed to next block}, \\
1 \le k_d \le (N-1)/2 & \longrightarrow & N^{d-1}(N-1)/2, \\
 & & \\
-(N-1)/2 \le k_{d-1} \le 1 & \longrightarrow & N^{d-2}(N-1)/2, \\
k_{d-1} = 0 & \longrightarrow & \text{proceed to next block}, \\
1 \le k_{d-1} \le (N-1)/2 & \longrightarrow & N^{d-2}(N-1)/2, \\
 & & \\
\ldots & \ldots & \ldots \\
 & & \\
-(N-1)/2 \le k_1 \le 1 & \longrightarrow & N^{0}(N-1)/2, \\
k_1 = 0 & \longrightarrow & 1, \\
1 \le k_1 \le (N-1)/2 & \longrightarrow & N^{0}(N-1)/2.
\end{array}$$

Adding up all the contributions one gets

$$\begin{aligned}
N^{d-1}(N-1) + N^{d-2}(N-1) + \ldots + (N-1) + 1 &= \\
(N-1)(N^{d-1} + N^{d-2} + \ldots + 1) + 1 &= \\
(N-1)\frac{N^d - 1}{N-1} + 1 &= \\
N^d - 1 + 1 &= N^d,
\end{aligned}$$

so that this case checks out. For even N we have, at each step of the algorithm,

$$\begin{array}{ccl}
-N/2+1 \leq k_d \leq 1 & \longrightarrow & 2^0 N^{d-1}(N/2-1), \\
k_d = 0 & \longrightarrow & \text{proceed to next block}, \\
1 \leq k_d \leq N/2-1 & \longrightarrow & 2^0 N^{d-1}(N/2-1), \\
k_d = N/2 & \longrightarrow & \text{proceed to next block}, \\
 & & \\
-N/2+1 \leq k_{d-1} \leq 1 & \longrightarrow & 2^1 N^{d-2}(N/2-1), \\
k_{d-1} = 0 & \longrightarrow & \text{proceed to next block}, \\
1 \leq k_{d-1} \leq N/2-1 & \longrightarrow & 2^1 N^{d-2}(N/2-1), \\
k_{d-1} = N/2 & \longrightarrow & \text{proceed to next block}, \\
 & & \\
\ldots & \ldots & \ldots \\
 & & \\
-N/2+1 \leq k_1 \leq 1 & \longrightarrow & 2^{d-1} N^0(N/2-1), \\
k_1 = 0 & \longrightarrow & 2^{d-1}, \\
1 \leq k_1 \leq N/2-1 & \longrightarrow & 2^{d-1} N^0(N/2-1), \\
k_1 = N/2 & \longrightarrow & 2^{d-1}.
\end{array}$$

The factors of powers of 2 that appear are due to the fact that there are two ways to proceed from each block to the next one. Adding up all the contributions one gets

$$\begin{aligned}
2^0 N^{d-1}(N-2) + 2^1 N^{d-2}(N-2) + \ldots + 2^{d-1} N^0(N-2) + 2^d &= \\
2^{d-1}(N-2)\left[\left(\frac{N}{2}\right)^{d-1} + \left(\frac{N}{2}\right)^{d-2} + \ldots + \left(\frac{N}{2}\right)^{0}\right] + 2^d &= \\
2^{d-1}(N-2)\frac{\left(\frac{N}{2}\right)^{d} - 1}{\frac{N}{2} - 1} + 2^d &= \\
2^d\left[\left(\frac{N}{2}\right)^{d} - 1\right] + 2^d &= \\
\left(N^d - 2^d\right) + 2^d &= N^d,
\end{aligned}$$

so that this case also checks out. Therefore, we have a completely well-defined algorithm that can be used to organize the independent variables, in any dimension. This algorithm should be used both when we calculate the Fourier components, in order to put them in their correct places in the conjugate lattice, and when one calculates an inverse transform or use the Fourier components in any other way, in order to get them from their correct places.

2 Indexing in Momentum Space

For the position-space representation $\varphi(\vec{n})$ of the field, with $1 \leq n_\mu \leq N$ and $\mu = 1, \ldots, d$, one can index the field into a single N-dimensional vector, using the single index ι instead of all the position coordinates, where

$$\iota = 1 + (n_1 - 1)N^0 + (n_2 - 1)N^1 + (n_3 - 1)N^2 + \ldots + (n_d - 1)N^{d-1}.$$

Defined in this way the index ι runs from 1 to N^d, thus enumerating all the sites of the lattice. There is also an algorithm to obtain the components of $\vec{n}$ back from the value of ι, as we will discuss later. We would like to do the same for the momentum-space representation $\widetilde{\varphi}(\vec{k})$, with $k_m \leq n_\mu \leq k_M$, $\mu = 1, \ldots, d$, where the extremes of the range are defined as

$$k_m = -(N-1)/2, \quad k_M = (N-1)/2,$$

for odd N, and as

$$k_m = -N/2 + 1, \quad k_M = N/2,$$

for even N. In order for the index value of 0 to correspond to the zero mode $\vec{k} = \vec{0}$, we would like to use the index κ given by

$$\kappa = k_1 N^0 + k_2 N^1 + k_3 N^2 + \ldots + k_d N^{d-1}.$$

In order to show that one can in fact do this, and to discover how to invert the relation, getting the components of $\vec{k}$ out of κ, we start with a version of the momentum-space index just like the position-space one,

$$\kappa' = 1 + (k_1 - k_m)N^0 + (k_2 - k_m)N^1 + (k_3 - k_m)N^2 + \ldots + (k_d - k_m)N^{d-1},$$

which ranges from 1 to N^d and where $0 \leq (k_\mu - k_m) \leq N - 1$, $\mu = 1, \ldots, d$, just like the position-space index. Now, starting from this index we have

$$\begin{aligned}
\kappa' &= 1 + k_1 N^0 + k_2 N^1 + k_3 N^2 + \ldots + k_d N^{d-1} \\
&\quad -(k_m N^0 + k_m N^1 + k_m N^2 + \ldots + k_m N^{d-1}) \\
&= k_1 N^0 + k_2 N^1 + k_3 N^2 + \ldots + k_d N^{d-1} \\
&\quad +1 - k_m (N^0 + N^1 + N^2 + \ldots + N^{d-1}) \\
&= k_1 N^0 + k_2 N^1 + k_3 N^2 + \ldots + k_d N^{d-1} + 1 - k_m \frac{N^d - 1}{N - 1},
\end{aligned}$$

where $N^d - 1$ is always divisible by $N - 1$, of course. Therefore, considering the definition of κ, we have

$$\kappa' = \left(1 - k_m \frac{N^d - 1}{N - 1}\right) + \kappa.$$

Since the two indices are related by an additive constant, either one can be used to index the modes. Since κ' ranges from 1 to N^d, we have for the extreme values of κ

$$(N^d-1)\frac{k_m}{N-1} = \kappa_m \leq \kappa \leq \kappa_M = (N^d-1)\left(1+\frac{k_m}{N-1}\right).$$

Note that k_m is negative, so that the lower extreme is negative. For odd N the extremes can be written explicitly as

$$-\frac{N^d-1}{2} \leq \kappa \leq \frac{N^d-1}{2},$$

and the range is therefore symmetrical around 0. Note that since N is odd so is N^d, and therefore N^d-1 is even and thus divisible by 2. For even N we get

$$-\frac{(N-2)(N^d-1)}{2(N-1)} \leq \kappa \leq \frac{N(N^d-1)}{2(N-1)},$$

so that in this case the range is not symmetrical, and there are $(N^d-1)/(N-1)$ more positive-index elements than negative-index elements. A little analysis will show that the integer divisions are exact in this case also, without any truncation: since N^d-1 is always divisible by $N-1$, we are left with $-(N-2)/2$ and $N/2$ to be considered, and since N is even both N and $N-2$ are divisible by 2.

If we recall our organization of real and imaginary parts as independent coordinates, as described in the previous section, it is interesting to observe that one can verify that for odd N the negative values of κ correspond to imaginary parts, the value 0 to the real mode, and the positive values to real parts. However, the same is *not* true for even N, due to the modes with one or more components equal to $N/2$, some of which have their imaginary parts at positive values of the index. Due to this, in general it will be necessary to implement an additional pair of indexing arrays in order to map real parts to the corresponding imaginary parts and vice-versa, whenever it becomes necessary to recover the two parts of one and the same mode.

It remains for us to discuss the inversion algorithm. The algorithm for extracting the position coordinates $\vec{n}$ from the position-space index ι works by integer division, with truncation, and in d dimensions is given by

$$\begin{array}{rcl}
\tau & = & \iota - 1, \\
n_d & = & 1 + \tau /\!\!/ N^{d-1}, \\
\tau & = & \tau - (n_d - 1)N^{d-1}, \\
n_{d-1} & = & 1 + \tau /\!\!/ N^{d-2}, \\
\tau & = & \tau - (n_{d-1} - 1)N^{d-2}, \\
n_{d-2} & = & 1 + \tau /\!\!/ N^{d-3}, \\
\tau & = & \tau - (n_{d-2} - 1)N^{d-3}, \\
\dots & \dots & \dots \\
n_2 & = & 1 + \tau /\!\!/ N^{1}, \\
\tau & = & \tau - (n_2 - 1)N^{1}, \\
n_1 & = & 1 + \tau /\!\!/ N^{0},
\end{array}$$

where τ is a temporary variable. It is therefore clear that we can use exactly the same algorithm for extracting the d shifted momentum coordinates $k_\mu - k_m + 1$ from the index κ'. Since κ and κ' are related by an additive constant, we can also use the algorithm for κ, so long as we start by adding to it the constant $1 - \kappa_m$, and substitute $n_\mu - 1$ by $k_\mu - k_m$ in the formulas, in order to get the components k_μ of $\vec{k}$, and therefore we have the algorithm

$$
\begin{array}{rcl}
\tau & = & \kappa - \kappa_m, \\
k_d & = & k_m + \tau /\!/ N^{d-1}, \\
\tau & = & \tau - (k_d - k_m) N^{d-1}, \\
k_{d-1} & = & k_m + \tau /\!/ N^{d-2}, \\
\tau & = & \tau - (k_{d-1} - k_m) N^{d-2}, \\
k_{d-2} & = & k_m + \tau /\!/ N^{d-3}, \\
\tau & = & \tau - (k_{d-2} - k_m) N^{d-3}, \\
\cdots & \cdots & \cdots \\
k_2 & = & k_m + \tau /\!/ N^1, \\
\tau & = & \tau - (k_2 - k_m) N^1, \\
k_1 & = & k_m + \tau /\!/ N^0.
\end{array}
$$

This completes the discussion of the indexing scheme in momentum space.

3 The Transform and its Inverse

Let us discuss now, in detail, how to implement numerically the finite Fourier transform and its inverse transform. The complete Fourier transform of a real scalar field is given by

$$\widetilde{\varphi}(\vec{k}) = \frac{1}{N^d} \sum_{\vec{n}} e^{\imath \frac{2\pi}{N} \vec{k} \cdot \vec{n}} \varphi(\vec{n}),$$

but we will only calculate and store the independent real and imaginary parts, $\widetilde{\varphi}(\vec{k}) = \mathfrak{R}(\vec{k}) + \imath \mathfrak{I}(\vec{k})$, which are given by

$$
\begin{array}{rcl}
\mathfrak{R}(\vec{k}) & = & \displaystyle \frac{1}{N^d} \sum_{\vec{n}} \cos\left(\frac{2\pi}{N} \vec{k} \cdot \vec{n} \right) \varphi(\vec{n}), \\
\mathfrak{I}(\vec{k}) & = & \displaystyle \frac{1}{N^d} \sum_{\vec{n}} \sin\left(\frac{2\pi}{N} \vec{k} \cdot \vec{n} \right) \varphi(\vec{n}).
\end{array}
$$

For use in the numerical programs we will implement these in matrix form, using a single (usually very large) $N^d \times N^d$ transformation matrix,

$$\widetilde{\varphi}_\kappa = \frac{1}{N^d} \sum_{\iota=1}^{N^d} F_{\kappa\iota} \varphi_\iota,$$

where $\widetilde{\varphi}_\kappa$ stands for either a real part or an imaginary part, depending on the value of the index κ, according to our classification of the independent variables in momentum space, and where, accordingly,

$$F_{\kappa\iota} = \cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right) \quad \text{or} \quad F_{\kappa\iota} = \sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right),$$

where κ corresponds to $\vec{k}$ and ι to $\vec{n}$ via the respective indexation schemes. Since $\vec{k}\cdot\vec{n}$ has a limited set of possible values, much smaller than the number of elements N^{2d} of $F_{\kappa\iota}$, and which will be used repeatedly for many different modes, in order to save both computation time and memory, we will calculate all necessary sine and cosine functions just once, beforehand, for future repeated use. We will store these values in a relatively small array $P(\tau)$ indexed by the integer $\tau = \vec{k}\cdot\vec{n}$, and construct a very large indexing matrix $\mathbb{I}_{\kappa\iota}$, of size $N^d \times N^d$, to pick up the necessary values of $P(\tau)$ for each combination of a mode in momentum space and a site in position space. Since this indexing matrix can be of 2-byte integers rather than of the 8-byte double-precision real numbers needed for the values of $P(\tau)$, in this way we save memory at the ratio of almost four to one. The range of the integer $\vec{k}\cdot\vec{n}$ is given by

$$d\,k_m N \leq (\vec{k}\cdot\vec{n}) \leq d\,k_M N,$$

so that the size of the small double-precision array is given by $d\,Nk_M - d\,Nk_m + 1$, where $k_M - k_m = N - 1$ for either odd or even N, so that the size of this array is $d\,N(N-1)+1$, of the order of $d\,N^2$, much smaller than N^{2d} for the values of d of most interest, from $d = 3$ to $d = 5$. When we deal with the values of $P(\tau)$ using this array, not all values along the array are actually going to be used, because not all integers in the range can in fact be written as $\vec{k}\cdot\vec{n}$. However, it can be verified that the rate of useful occupation of the array is large for the dimensions of interest, in fact, it is over 90 % for dimensions from $d = 3$ to $d = 5$. Therefore, we have here only a very small waste of memory space, and in the interest of simplicity it is not worth while to try to improve this occupation rate.

Since we must store the values of both the sines and the cosines, we actually have to double the size of the array as it was described above. In order to put the cosines at positive values of the index and the sines at negative values, we add and subtract constants in order to use the ranges that follow,

$$-d\,N(N-1) - 1 \leq \left[\vec{k}\cdot\vec{n} - (d\,Nk_M + 1)\right] \leq -1,$$

to be used for the sines, and

$$1 \leq \left[\vec{k}\cdot\vec{n} + (-d\,Nk_m + 1)\right] \leq d\,N(N-1) + 1,$$

to be used for the cosines, so that the total range is given by

$$-d\,N(N-1) - 1 = \tau_m \leq \tau \leq \tau_M = d\,N(N-1) + 1,$$

The total size of the array is now given by $2d\,N(N-1)+3$, which is still much smaller than the indexing matrix. The indexing matrix $\mathbb{I}_{\kappa\iota}$ will return the values of τ in this range, and is to be used as the argument of $P(\tau)$, given values of κ and ι, so that we have

$$F_{\kappa\iota} = P(\mathbb{I}_{\kappa\iota}),$$

and the transformation can now be written as

$$\widetilde{\varphi}_\kappa = \frac{1}{N^d} \sum_{\iota=1}^{N^d} P(\mathbb{I}_{\kappa\iota})\varphi_\iota,$$

which includes all the independent real and imaginary parts, depending on the values of κ, in a single matrix operation, which is easy to implement efficiently.

Let us now examine the situation concerning the inverse transform. It is clear that a similar scheme should be used for it. Since the indexing matrix $\mathbb{I}_{\kappa\iota}$ is so large, and considering the possibility that a single program might need to use both the transform and its inverse, it is important to build the scheme for the inverse transform in such a way that it is able to use the same indexing matrix. As we will see, it is possible to do this, but unfortunately not in a very efficient way. For speed of execution the order of the indices matters, and should in fact be $\mathbb{I}_{\iota\kappa}$ for the direct transform and $\mathbb{I}_{\kappa\iota}$ for the inverse transform. Hence, one can save memory only so long as one needs only one of the two matrices or so long as the program is not dependent on the performance of one of the two transforms. The inverse Fourier transform is defined by

$$\varphi(\vec{n}) = \sum_{\vec{k}} e^{-\imath\frac{2\pi}{N}\vec{k}\cdot\vec{n}}\widetilde{\varphi}(\vec{k}),$$

or, in N^d-dimensional matrix form,

$$\varphi_\iota = \sum_{\kappa=\kappa_m}^{\kappa_M} F^{-1}_{\iota\kappa}\widetilde{\varphi}_\kappa,$$

where the limits of the sum over the momentum index are $\kappa_m = (N^d-1)k_m/(N-1)$ and $\kappa_M = (N^d-1)(1+k_m/(N-1))$. We must now deal explicitly with the fact that both the Fourier component of the field and the mode function are complex. Since we have the decomposition of the field $\widetilde{\varphi}(\vec{k}) = \mathfrak{R}(\vec{k}) + \imath\mathfrak{I}(\vec{k})$, as well as

$$e^{-\imath\frac{2\pi}{N}\vec{k}\cdot\vec{n}} = \cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right) - \imath\sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right),$$

it follows that we have

$$\begin{aligned}
\varphi(\vec{n}) &= \sum_{\vec{k}}\left[\cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}) + \sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{I}(\vec{k})\right] \\
&\quad +\imath\sum_{\vec{k}}\left[-\sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}) + \cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{I}(\vec{k})\right].
\end{aligned}$$

Since the field $\varphi(\vec{n})$ is real, the sum giving its imaginary part must vanish, and we are left with

$$\varphi(\vec{n}) = \sum_{\vec{k}} \left[\cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}) + \sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{I}(\vec{k})\right].$$

We must now consider separately the contributions to the sum from the real and complex modes. From the purely real modes we get the contributions

$$\sum_{\text{real } \vec{k}} \cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}),$$

while from the complex modes we get the contributions

$$\sum_{\text{complex } \vec{k}} \left[\cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}) + \sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{I}(\vec{k})\right].$$

Since we keep only one copy of each independent $\mathfrak{R}(\vec{k})$ and one copy of each independent $\mathfrak{I}(\vec{k})$, while the sum above runs over both parts on all the complex modes, in the case of these modes we must multiply the contribution by a factor of two. In this way we get the correct result when we run over the modes and use only the part which is stored in each one. So in reality we will calculate this contribution as

$$\sum_{\text{complex } \vec{k}} \left[2\cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}) + 2\sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{I}(\vec{k})\right].$$

These factors of two, which do not appear in the real modes, prevent us from implementing the inverse transform as a single matrix operation. The best way to implement it will be to first perform a complete matrix operation with the factors of two for all modes, and then subtract back once the contributions of the real modes,

$$\begin{aligned}\varphi(\vec{n}) &= 2\sum_{\vec{k}} \left[\cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}) + \sin\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{I}(\vec{k})\right] \\ &\quad - \sum_{\text{real } \vec{k}} \cos\left(\frac{2\pi}{N}\vec{k}\cdot\vec{n}\right)\mathfrak{R}(\vec{k}).\end{aligned}$$

This second sum runs over only either $\varrho_M = 1$ or $\varrho_M = 2^d$ elements, depending on the parity of N, and is therefore much smaller than the first one, which runs over N^d elements, specially for large lattices, representing therefore only a small additional computational effort. In order to do this it will be necessary to construct a small indexing array to pick the purely real modes out of the indexed vector of modes. Note that the first term of the inverse transformation matrix $F^{-1}_{\iota\kappa}$, as decomposed above, is equal to $2F_{\kappa\iota}$. In order to deal with the second term, consider an indexing array $\lambda(\varrho)$ of dimension ϱ_M, which for each $\varrho = 1, \ldots, \varrho_M$, with $\varrho_M = 1$ for odd N

and $\varrho_M = 2^d$ for even N, returns the value of the index κ of a purely real mode. Using it we can write the inverse transform as

$$\varphi_\iota = 2 \sum_{\kappa=\kappa_m}^{\kappa_M} P(\mathbb{I}_{\kappa\iota})\widetilde{\varphi}_\kappa - \sum_{\varrho=1}^{\varrho_M} P(\mathbb{I}_{\lambda(\varrho)\iota})\widetilde{\varphi}_{\lambda(\varrho)}.$$

We have therefore the inverse transformation implemented by a large matrix operation followed by a small correction, which is still easy to implement efficiently. In the computer code we will actually have two indexation matrices $\mathbb{I}^{(t)}_{\kappa\iota}$ and $\mathbb{I}^{(i)}_{\iota\kappa} = \mathbb{I}^{(t)}_{\kappa\iota}$, differing only by the ordering of the indices, one for the direct transformation and one for the inverse transformation, so that both can be executed efficiently.

4 Fortran Routines

The technique described above for the implementation of the Fourier transforms implies strong limitations for the lattice size, both due to addressing issues and due to memory requirement issues. We will examine here these limitations, and we will see that the technique is still usable for lattices with the usual sizes in the important cases $d = 3$ and $d = 4$. For these lattices a set of routines implementing the transforms in the fastest possible way is supplied. For larger lattices a different set of routines is supplied, in which the indexation matrices $\mathbb{I}^{(t)}_{\kappa\iota}$ and $\mathbb{I}^{(i)}_{\iota\kappa}$ are not implemented directly in memory, but rather by functions that dynamically calculate and return the index τ of the phases, appropriately shifted, given values of the coordinate indices κ and ι. These routines are about 4 to 5 times slower, but can be used for very large lattices, with much smaller memory requirements.

First of all, note that since the indexation matrices are defined as 2-byte signed integers, they are limited to the range $[-32767, 32767]$, that is, to 65535 different values. Therefore this last number is the maximum size of the indexed array of phases $P(\tau)$. This implies that there are limits to the possible lattice sizes N in each dimension d, as shown in the following table:

d	$N_{\max}$	$\mathrm{RAM}_{\max}$
1	181	64 kB
2	128	520 MB
3	105	2497 GB
4	91	8560 TB
5	81	22 EB

Here the corresponding maximum required amounts of RAM are also shown and the order of the unit multipliers is k, M, G, T, P and E, ranging from 2^{10} to 2^{60}, so we see that in the case of the higher dimensions the memory requirements are truly enormous for the larger lattices still allowed by this indexing limitation. Specially for dimensions $d = 4$ and $d = 5$, these are larger lattices than one can possibly expect to run successfully on current computers. On 32-bit processors there is another type of limitation on the values of N, which is far more serious for the larger values of d. The limitations of the signed 32-bit integer addressing of the indexation matrices limits their size, and hence the size of the lattice, as shown in the following table:

d	N_{max}	$\mathrm{RAM}_{\mathrm{max}}$
1	32767	2 GB
2	181	2 GB
3	31	2 GB
4	13	2 GB
5	7	2 GB

In this case the addressing limitations correspond to a fixed maximum required memory, of course. In addition to this, the limitations of the signed 32-bit integer addressing of the indexed array of phases also limits its size and the size of the lattice, but this one is a significant limitation only for the case $d = 1$, which is relatively irrelevant to us, as shown in the following table:

d	N_{max}	$\mathrm{RAM}_{\mathrm{max}}$
1	11585	2 GB
2	8192	2 GB
3	6689	2 GB
4	5793	2 GB
5	5181	2 GB

Therefore, we see that this last limitation can be safely ignored. The corresponding limitations in the case of a 64-bit processor with 48-bit addressing are smaller, and not so significant, since in this case the memory availability limitations become relevant much before the addressing limitations themselves. The limits due to the 48-bit addressing of the indexation matrices are shown in the following table:

d	N_{max}	$\mathrm{RAM}_{\mathrm{max}}$
1	8388607	128 TB
2	2896	128 TB
3	203	128 TB
4	53	128 TB
5	24	128 TB

The limits due to the 48-bit addressing of the indexed array of phases are shown in the following table:

d	N_{max}	$\mathrm{RAM}_{\mathrm{max}}$
1	2965821	128 TB
2	2097152	128 TB
3	1712317	128 TB
4	1482910	128 TB
5	1326355	128 TB

If we consolidate all these indexing and addressing limitations in a single table, we see that the limits on a 32-bit processor with 32-bit logical addressing are given by the table below:

d	$N_{\max}$	$\mathrm{RAM}_{\max}$
1	181	64 kB
2	128	520 MB
3	31	2 GB
4	13	2 GB
5	7	2 GB

The same limits on a 64-bit processor with 48-bit logical addressing are given by the table below:

d	$N_{\max}$	$\mathrm{RAM}_{\max}$
1	181	64 kB
2	128	520 MB
3	105	2497 GB
4	53	128 TB
5	24	128 TB

A typical larger lattice in $d = 4$ would have $N = 10$, and in this case the transform can be used with reasonable memory requirements, about 200 MB for a single indexation matrix, and about 400 MB for two indexation matrices. A lattice of $N = 20$ in $d = 3$ can be used with memory requirements of about 128 MB for one matrix and of about 256 MB for two matrices. Therefore, we conclude that the fast version of the transforms can be used in practice in most simulations. Whenever there is not enough memory available, one can turn to the alternative routines.

There are Fortran routines available implementing the direct and inverse Fourier transforms, as described in this paper. There are also a few test programs that may be useful as examples of how to use the routines. As they currently stand, these routines are written for use in either 32-bit or 64-bit processors. However, sufficiently large lattices may require a 64-bit processor, as well as lots of available memory. The files described in what follows, containing the source code, are freely available in a compressed tar file at the URL:

`http://latt.if.usp.br/technical-pages/fmrsf/`

There are files with the definition of the necessary data structures, as well as the initialization routines and the transforms themselves. These modules are meant to be integrated into larger programs at a source-code level. The files containing the data structures are include files, meant to be included in the modules that need access to the data structures they contain. All interchange of data among modules is made by means of common blocks. Each initialization routine defines the data in the common block in the corresponding include file. These initialization routines should be called once at the beginning of the program, one for each common block that is used in the program. Read the makefile or the source code in order to see which modules depend on which data structures.

Here are short explanations of the nature of each source-code file. First the files with the data structures:

`fp_def.f`: an include file with the basic parameters defining the run; this is needed by all modules.

`fp_cal.f`: an include file with other parameters, which are calculated using the basic ones; this is needed by most modules.

`cb_param.f`: an include file with a common block containing additional calculated parameters.

`cb_flags.f`: an include file with a common block containing the dimensionality flags.

`cb_logic.f`: an include file with a common block containing the logical array for marking the sites as carrying either their real parts or their imaginary parts.

`cb_field.f`: an include file with a common block containing the field components in position space and in momentum space.

`cb_phase.f`: an include file with a common block containing the indexed array for the phases.

`cb_trans.f`: an include file with a common block containing the large indexing matrix for the direct transform.

`cb_trinv.f`: an include file with a common block containing the large indexing matrix for the inverse transform.

`cb_traux.f`: an include file with a common block containing the auxiliary indexing array for the inverse transform.

The files containing the main routines, including the initialization routines and the transforms themselves:

`check_pars.f`: a subroutine that verifies consistency of the basic parameters defining the run.

`check_pars_32.f`: a subroutine that verifies consistency of the basic parameters regarding indexation and addressing issues on a 32-bit machine with 32-bit addressing capability.

`check_pars_48.f`: a subroutine that verifies consistency of the basic parameters regarding indexation and addressing issues on a 64-bit machine with 48-bit addressing capability.

`init_param.f`: a subroutine that initializes the common block in `cb_param.f`.

`init_flags.f`: a subroutine that initializes the common block in `cb_flags.f`.

`init_logic.f`: a subroutine that initializes the common block in `cb_logic.f`.

`init_phase.f`: a subroutine that initializes the common block in `cb_phase.f`.

`init_trans.f`: a subroutine that initializes the common block in `cb_trans.f`.

`init_trinv.f`: a subroutine that initializes the common block in `cb_trinv.f`.

`init_traux.f`: a subroutine that initializes the common block in `cb_traux.f`.

`indphtrn.f`: a function that replaces the large indexing matrix in the common block in `cb_trans.f`.

`indphinv.f`: a function that replaces the large indexing matrix in the common block in `cb_trinv.f`.

`fourier_trans_fast.f`: a subroutine that implements the fast version of the direct Fourier transform.

`fourier_inver_fast.f`: a subroutine that implements the fast version of the inverse Fourier transform.

`fourier_trans_large.f`: a subroutine that implements the direct Fourier transform with much smaller memory requirements, for use on large lattices.

`fourier_inver_large.f`: a subroutine that implements the inverse Fourier transform with much smaller memory requirements, for use on large lattices.

`fourier_trans_slow.f`: a subroutine that implements the fast version of the direct Fourier transform in a much slower but somewhat less memory-intensive way.

`fourier_inver_slow.f`: a subroutine that implements the fast version of the inverse Fourier transform in a much slower but somewhat less memory-intensive way.

Files containing the test programs and some auxiliary routines, including a couple of C interfaces to system facilities:

`test_trans_inver_fast.f`: a program that composes the fast versions of the transform and of the inverse, in this order, and measures the resulting average error.

`test_inver_trans_fast.f`: a program that composes the fast versions of the inverse and of the transform, in this order, and measures the resulting average error.

`test_trans_inver_large.f`: a program that composes the large-lattice versions of the transform and of the inverse, in this order, and measures the resulting average error.

`test_inver_trans_large.f`: a program that composes the large-lattice versions of the inverse and of the transform, in this order, and measures the resulting average error.

`test_trans_inver_slow.f`: a program that composes the slow versions of the transform and of the inverse, in this order, and measures the resulting average error.

`test_inver_trans_slow.f`: a program that composes the slow versions of the inverse and the transform, in this order, and measures the resulting average error.

`urandom_ctof.c`: a Fortran-callable C interface to the system libraries, to get a random seed from the Linux kernel.

`clock_ctof.c`: a Fortran-callable C interface to the system libraries, to get the CPU time of the current process from the system.

`dranr-1.f`: the first random number generator from the authors of the book "Numerical Recipes".

`dranr-2.f`: the second random number generator from the authors of the book "Numerical Recipes".

`sdot.f`: a simple function that returns the scalar product of two vectors.