European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS 2004
P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer (eds.)
Jyväskylä, 24–28 July 2004

GENERIC PROGRAMMING ASPECTS OF SYMMETRY EXPLOITING NUMERICAL SOFTWARE

Malin Ljungberg* and Krister Åhlander*

*Department of Information Technology
Uppsala University, Box 337, S-751 05 Uppsala, Sweden
e-mails: Malin.Ljungberg@it.uu.se,Krister.Ahlander@it.uu.se

Key words: Numerical Software, Generic Programming, Generalized Fourier Transform, Boundary Element Method, Representation Theory.

Abstract. *The use of the generalized Fourier transform as a means to diagonalize certain types of equivariant matrices, and thus speeding up the solution of numerical systems, is discussed. Such matrices may arise in various applications with geometrical symmetries, for example when the boundary element method is used to solve an electrostatic problem in the exterior of a symmetric object. The method is described in detail for an object with a triangular symmetry, and the feasibility of the method is confirmed by numerical experiments.*

The design of numerical software for this kind of applications is a challenge. It is argued that generic programming is very suitable in this context, mainly because it is type safe and promotes polymorphism capabilities in link time.

A generic C++ design of important mathematical abstractions such as groups, vector spaces, and group algebras, is outlined, illustrating the potential provided by generative programming techniques. The integration of explicit support for various data layouts for efficiency tuning purposes is discussed.

1 INTRODUCTION

The mathematical notion of a group is an example of an abstraction which does not conveniently fit into the frame of matrices and linear algebra, which are the most commonly used abstractions in the field of numerical PDE (partial differential equation) solvers. There are several interesting applications where groups and the understanding of groups may have a substantial impact. For example, PDEs which stem from applications which exhibit domain symmetries can utilize groups for describing the domain symmetries. This abstract description yields a structured way of exploiting the symmetries in order to reduce memory requirements and numerical complexity. The method we consider in this paper is based on a block diagonalization of a system matrix via a generalized Fourier transform. It has been pioneered by Allgower and others [1, 2], and is discussed thoroughly in a forthcoming paper by Åhlander and Munthe-Kaas [3]. A similar approach is also described in [4]. For an overview on generalized Fourier transforms, we recommend [5, 6]. However, implementation aspects of this theory applied to numerical computations are less well developed.

The design and implementation of software abstractions based upon group theoretical axioms is therefore in focus in our present project. Specifically, we consider boundary element applications for domains with symmetries, and we use the generalized Fourier transform to block diagonalize the resulting system. Generative programming [7] is used in order to address generality, type safety, flexibility, and performance issues. Generative programming decouples data structures from algorithms by means of iterators, which is demonstrated for example by the standard template library [8]. By carefully specifying the group interface, which besides the obvious group operation, inverse, and identity also supplies means for traversal, input, output etc., we achieve reusable implementations of mathematical groups with strong type checking. Furthermore, since polymorphism is supported in link time, unnecessary runtime overhead is avoided.

The outline of this paper is as follows. In Section 2, we give an overview of the boundary element method and we introduce some important concepts from group theory. It is explained how groups representing domain symmetries may be exploited to reduce memory requirements and computations via a generalized Fourier transform. In Section 3, we illustrate the approach by means of an example. In Section 4, we focus on how generative programming can be used to design the required mathematical abstractions in a type safe and generic manner. We conclude, in Section 5, that generic programming is promising for this area.

2 BACKGROUND

2.1 Boundary element methods

A homogenous differential equation can be reformulated as an integral equation on the boundary. When we use a standard finite element method to solve this integral equation numerically, we obtain the boundary element method, BEM. In this section, we use an

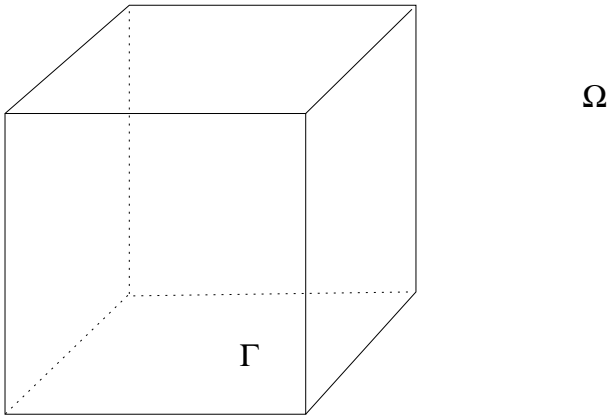


Figure 1: A computational domain with cubic symmetry.

example from electrostatics to illustrate the method. See e.g. [9] for more examples.

Consider the electric potential u outside a body with boundary Γ, and let Ω denote the exterior, see Figure 1. If the electric potential is given on Γ, the PDE to be solved is the following:

$$\begin{array}{rcll} \triangle u(x) & = & 0 & x \in \Omega \\ u(x) & = & u_0(x) & x \in \Gamma \end{array} \tag{1}$$

This external Dirichlet problem may be numerically computed by BEM. First, we reformulate the problem as an integral equation. We note that the electric potential around a charge Q situated at a point y is given by

$$u(x) = \frac{Q}{4\pi|x-y|}, \ x \neq y.$$

By assuming that the charge is situated on Γ and that the charge per area element on Γ is given by an unknown function q, we may deduce that the solution to (1) is given by

$$u(x) = \frac{1}{4\pi} \int_\Gamma \frac{q(y)}{|x-y|} d\gamma(y), \ x \in \Omega, \tag{2}$$

where $d\gamma(y)$ is an element of surface area at position y. By letting $x \to y$ and the fact that u is continuous, we find that the same integral holds on the boundary Γ where $u(x) = u_0(x)$:

$$\frac{1}{4\pi} \int_\Gamma \frac{q(y)}{|x-y|} d\gamma(y) = u_0(x), \ x \in \Gamma. \tag{3}$$

This integral is known as a Fredholm integral of the first kind, and may be written

$$Kq = u_0,$$

where K is the appropriate integral operator. It has a weak discontinuity but it is well defined and techniques exist for computing it numerically [10].

Second, we need to solve the integral equation $Kq = u_0$ numerically. Applying the finite element method, we discretize Γ into n discrete elements $\mathcal{T}_h = \{T_i\}_{i=1}^n$ such that $\Gamma = \bigcup_{i=1}^n T_i$. For simplicity, we consider the (discontinuous) space of functions which are constant on each T_i. A basis of this space, denoted by W_h, is given by $\{\psi_j\}_{j=1}^n$ where

$$\psi_j(x) = \begin{cases} 1, & x \in T_j \\ 0, & x \notin T_j \end{cases} .$$

By multiplying (3) with a test function p and integrating over Γ, we may restate the integral as

$$\int_\Gamma Kq(x)p(x)d\gamma(x) = \int_\Gamma u_0(x)p(x)d\gamma(x).$$

Approximating $q(x)$ as $q^h(x) = \sum_{j=1}^n \xi_j\psi_j(x) \in W_h$, and assuming that the equation holds for all $p \in W_h$—and thus for each $\psi_i(x)$—we obtain

$$\int_\Gamma Kq^h(x)\psi_i(x)d\gamma(x) = \int_\Gamma u_0(x)\psi_i(x)d\gamma(x),\ i = 1\ldots n.$$

By simplifying and reordering, we obtain the following relation for $i = 1\ldots n$:

$$\begin{aligned} \int_\Gamma \frac{1}{4\pi}\int_\Gamma \sum_{j=1}^M \xi_j\psi_j(y)\psi_i(x)d\gamma(x) &= \sum_{j=1}^n \frac{1}{4\pi}\int_{T_i}\int_{T_j}\frac{1}{|x-y|}d\gamma(y)d\gamma(x)\xi_j = \\ \int_\Gamma u_0(x)\psi_i(x)d\gamma(x) &= \int_{T_i} u_0(x)d\gamma(x). \end{aligned} \tag{4}$$

This is a linear system of equations in the n unknowns ξ_j, and we may rewrite (4) as

$$\mathbf{Ax} = \mathbf{f}, \tag{5}$$

where the unknown $\mathbf{x}$ is the column vector $(\xi_1, \ldots, \xi_n)^T$, the right hand side entries are

$$f_i = \int_{T_i} u_0(x)d\gamma(x)$$

and the matrix entries are

$$\mathbf{A}_{i,j} = \frac{1}{4\pi}\int_{T_i}\int_{T_j}\frac{1}{|x-y|}d\gamma(y)d\gamma(x). \tag{6}$$

Note that $\mathbf{A}$ is dense, and the cost for computing the solution of (5) with a direct method is in general $\approx \frac{2}{3}n^3$. We acknowledge that other, faster, methods exist for solving boundary integral equations, which utilize the fact that $\mathbf{A}$ is rank deficient on off diagonal blocks [11]. We focus, however, on another property of $\mathbf{A}$, in the case when Γ has symmetries. In this case, many entries of $\mathbf{A}_{i,j}$ will be identical. In order to understand and exploit this property systematically, we need some additional mathematical machinery.

2.2 Group theoretical background

The mathematical concept of a group is the adequate abstraction for formalizing domain symmetries. In this section we will give a brief overview of some group theoretical notions which are useful for our applications. For more detailed information concerning groups, representations and generalized Fourier transforms, we refer to e.g. [12, 13].

A set of elements $\mathcal{G}$ together with a binary operation $g, h \mapsto gh$ is called a *group* if the operation is associative, if there is an identity e in the group, and if each group element g has an inverse g^{-1}. That is, for each $f, g, h \in \mathcal{G}$, we have $(fg)h = f(gh)$, $ge = eg = g$, and $gg^{-1} = g^{-1}g = e$. We will consider only finite groups for which $|\mathcal{G}| < \infty$.

In order to utilize a group when representing objects with certain domain symmetries, the group algebra is useful. The *group algebra* $\mathbb{C}\mathcal{G}$ is a vector space $\mathbb{C}^{|\mathcal{G}|}$ equipped with a multiplication $* : \mathbb{C}\mathcal{G} \times \mathbb{C}\mathcal{G} \to \mathbb{C}\mathcal{G}$. We choose $g \in \mathcal{G}$ as a basis for $\mathbb{C}\mathcal{G}$, so for $x \in \mathbb{C}\mathcal{G}$, $x = \sum_{g\in\mathcal{G}} x(g)g$. For $x, y \in \mathbb{C}\mathcal{G}$ we define $x * y$ via the convolution:

$$(x * y)(g) = \sum_{h\in\mathcal{G}} x(gh)y(h^{-1}). \tag{7}$$

We also need some basic notions from representation theory. A representation ρ is a map $\rho : \mathcal{G} \to \mathbb{C}^{d\times d}$ such that $\rho(gh) = \rho(g)\rho(h)$ for all $g, h \in \mathcal{G}$. The number d is called the dimension of ρ and is denoted $\dim \rho$. Note that a similarity transform of a representation yields another representation, since $T\rho(gh)T^{-1} = T\rho(g)\rho(h)T^{-1} = (T\rho(g)T^{-}1)(T\rho(h)T^{-1})$ for any non-singular $T \in \mathbb{C}^{d\times d}$. Two representations $g \mapsto \rho(g)$ and $g \mapsto T\rho(g)T^{-1}$ are said to be *isomorphic*.

A representation is said to be *reducible*, if it is possible to block diagonalize it via a similarity transform. If this is not possible, the representation is said to be *irreducible*. A remarkable fact from representation theory is that for each finite group, there exists a finite set of non isomorphic irreducible representations, which we denote $\mathcal{R} = \{\rho_0, \ldots, \rho_{q-1}\}$ and we let $d_r = \dim \rho_r$. For most groups of mathematical physics, it is easy to find $\mathcal{R}$ in the literature, see e.g. [14].

In our context, representations are important because $\mathcal{R}$ can be used to transform entities in the group algebra $\mathbb{C}\mathcal{G}$ to an isomorphic block diagonal matrix algebra $\widehat{\mathbb{C}\mathcal{G}} = \prod_{r=0}^{q-1} \mathbb{C}^{d_r\times d_r}$. Thus, $\hat{x} \in \widehat{\mathbb{C}\mathcal{G}}$ is a block diagonal matrix whose rth block has dimension $d_r \times d_r$. As a notational convenience, we let $\hat{x}(\rho_r)$ denote the rth block of $\hat{x}$. The generalized Fourier transform, $\text{gft} : \mathbb{C}\mathcal{G} \to \widehat{\mathbb{C}\mathcal{G}}$, is given by

$$\hat{x}(\rho_r) = \sum_{g\in\mathcal{G}} x(g)\rho_r(g). \tag{8}$$

The inverse transform, $\text{igft} : \widehat{\mathbb{C}\mathcal{G}} \to \mathbb{C}\mathcal{G}$, is given by

$$y(g) = \frac{1}{|\mathcal{G}|} \sum_{r=0}^{q-1} d_r \text{trace}(\rho_r(g^{-1})\hat{y}(\rho_r)). \tag{9}$$

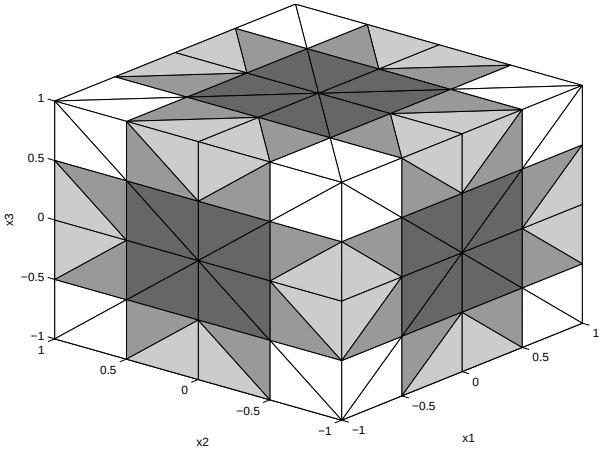


Figure 2: A symmetry respecting discretization of the cube boundary. Elements of same shade of gray belong to the same orbit.

Since the generalized Fourier transform is an algebra isomorphism, we have $\mathrm{igft}(\mathrm{gft}(x)) = x$ and $\mathrm{gft}(x * y) = \mathrm{gft}(x)\mathrm{gft}(y)$. We remark that the usual discrete Fourier transform is just a special case of the generalized Fourier transform. In this case, the underlying group is cyclic: $\mathcal{C}_n = \langle a \mid a^n = e \rangle$. (A cyclic group of order n is isomorphic to the set of integers $\{0, \ldots, n-1\}$ when addition modulo n is used as the group operation, cf. Section 4.1 where an implementation of $\mathcal{C}_n$ is discussed.)

2.3 Using groups to exploit domain symmetries for BEM

We can now describe how group theory can help us to reduce computations for BEM when the domain has symmetries, for instance as in Figure 1. First, we note the connection between domain symmetries and groups: The set of linear transformations which map Γ onto itself forms a group under the binary operation of function composition. It is clear that it is a group, because function composition is associative, the identity transformation is the identity of the group and each transformation has a unique inverse. In the case of the symmetries of the cube, this group has 48 elements, see e.g. [12]. We remark that every transformation is a reflection or a rotation and it is thus isometric.

Next, we assume that the domain is discretized with a symmetry respecting discretization, see Figure 2. This means that for each transformation $g \in \mathcal{G}$, each element T_i in the triangularization $\mathcal{T}_h$ is mapped onto another element $T_j = g(T_i) \in \mathcal{T}_h$. We may use this observation to define a (right) action $i, g \mapsto ig$ of the group $\mathcal{G}$ on the indices $\mathcal{I}$ via the relation

$$T_{ig} = g(T_i). \tag{10}$$

Returning to equation (5), we notice that if we use $g \in \mathcal{G}$ to change our variables of integration in (6) and the fact that g is isometric, we obtain

$$\mathbf{A}_{i,j} = \mathbf{A}_{ig,jg}. \tag{11}$$

A matrix $\mathbf{A}$ with this property is called equivariant under $\mathcal{G}$, and this property is the key to efficiently representing and computing the entries of $\mathbf{A}$.

In order to describe how to efficiently store $\mathbf{A}$, we first note that the group action partitions $\mathcal{I}$ into separate *orbits*. The orbit of an index i is $\mathcal{O}(i) = \{j \in \mathcal{I} \mid j = ig \text{ for some } g \in \mathcal{G}\}$. By picking one index s from each orbit, we obtain a selection $\mathcal{S} \subset \mathcal{I}$ which represents the set of orbits. The number of distinct orbits is $m = |\mathcal{S}|$. For each $i \in \mathcal{I}$ we can find $s \in \mathcal{S}$ and $g \in \mathcal{G}$ such that $i = sg$.

If the action is such that each orbit contains $|\mathcal{G}|$ elements, the action is said to be *free*. In this case, each $s \in \mathcal{S}$ and $g \in \mathcal{G}$ are uniquely defined by $sg = i$ for each $i \in \mathcal{I}$. This property makes the theory for exploiting domain symmetries particularly simple, and henceforth we assume that the action is free. General actions are treated in e.g. [1, 2, 3].

Since we assume the action to be free, we can use $s \in \mathcal{S}$ and $g \in \mathcal{G}$ to index elements in $\mathbb{C}^n$. We use the notation $x_s(g) = \mathbf{x}_i$, and denote with $\mathbb{C}^m\mathcal{G}$ the space indexed with $\mathcal{S} \times \mathcal{G}$. Similarly, we can index $\mathbf{A}$ with four indices, $s, t \in \mathcal{S}$ and $g, h \in \mathcal{G}$. But due to the equivariance property (11) we have $\mathbf{A}_{sg,th} = \mathbf{A}_{sgh^{-1},t}$, and we represent $\mathbf{A}$ as

$$A_{s,t}(g) = \mathbf{A}_{sg,t}.$$

When the action is free, $n = m|\mathcal{G}|$. We are thus able to reduce the memory requirements for $\mathbf{A}$ from $n^2 = m^2|\mathcal{G}|^2$ to $m^2|\mathcal{G}|$, a reduction which may be significant.

We denote with $\mathbb{C}^{m\times m}\mathcal{G}$ the vector space indexed with $\mathcal{S} \times \mathcal{S} \times \mathcal{G}$. We now generalize the convolution (7) to a "block" version. We define multiplication $* : \mathbb{C}^{m\times n}\mathcal{G} \times \mathbb{C}^{n\times p}\mathcal{G} \to \mathbb{C}^{m\times p}\mathcal{G}$ for $x \in \mathbb{C}^{m\times n}\mathcal{G}$ and $y \in \mathbb{C}^{n\times p}\mathcal{G}$ via

$$(x * y)_{i,k}(g) = \sum_{j\in\mathcal{S},h\in\mathcal{G}} x_{i,j}(gh^{-1})y_{j,k}(h). \tag{12}$$

Returning to (5), we can now see how domain symmetry is exploited. First, we represent $\mathbf{A}$ more efficiently, and second, we replace the matrix vector multiplication with the block convolution.

Proposition 1 *For $\mathbf{A}$ equivariant under a free action of $\mathcal{G}$, $\mathbf{Ax}$ can be computed via $A*x$ where $A \in \mathbb{C}^{m\times m}\mathcal{G}$ represents $\mathbf{A}$ and $x \in \mathbb{C}^m\mathcal{G} = \mathbb{C}^{m\times 1}\mathcal{G}$ equals $\mathbf{x}$.*

Proof For $i = sg$, we have

$$(\mathbf{Ax})_i \;=\; \sum_{j\in\mathcal{I}} \mathbf{A}_{i,j}\mathbf{x}_j = \sum_{t\in\mathcal{S},h\in\mathcal{G}} \mathbf{A}_{i,th}\mathbf{x}_{th} = \sum_{t\in\mathcal{S},h\in\mathcal{G}} A_{s,t}(gh^{-1})x_t(h) = (A * x)_s(g).$$

We remark that our notation is chosen to make it simple to express the block convolution in alternative ways. We identify $\mathbb{C}^{m\times n}\mathcal{G}$ with $\mathbb{C}^{m\times n} \otimes \mathbb{C}\mathcal{G}$. For $x \in \mathbb{C}^{m\times n}\mathcal{G}$, we have $x(g) \in \mathbb{C}^{m\times n}$ and $x_{i,j} \in \mathbb{C}\mathcal{G}$. The convolution (12) may thus be expressed as a matrix product of convolutions,

$$(x * y)_{ik} = \sum_{j\in\mathcal{S}} x_{i,j} * y_{j,k}, \tag{13}$$

or as a convolution of matrix products,

$$(x * y)(g) = \sum_{h \in \mathcal{G}} x(gh)y(h^{-1}). \tag{14}$$

Finally, we lift the generalized Fourier transform and its inverse to block versions. Let $\text{gft} : \mathbb{C}^{m\times n} \otimes \mathbb{C}\mathcal{G} \to \mathbb{C}^{m\times n} \otimes \widehat{\mathbb{C}\mathcal{G}}$ and $\text{igft} : \mathbb{C}^{m\times n} \otimes \widehat{\mathbb{C}\mathcal{G}} \to \mathbb{C}^{m\times n} \otimes \mathbb{C}\mathcal{G}$ be given by

$$\hat{x}_{i,j} = \text{gft}(x_{i,j}) \tag{15}$$
$$y_{i,j} = \text{igft}(\hat{y}_{i,j}). \tag{16}$$

In summary, when $\mathbf{A}$ is equivariant w.r.t. a free group action, the equivariance can be exploited to solve $\mathbf{Ax} = \mathbf{f}$ more efficiently as follows:

1. Represent $\mathbf{A}$ and $\mathbf{f}$ as $A \in \mathbb{C}^{m\times m}\mathcal{G}$ and $f \in \mathbb{C}^m\mathcal{G}$.
2. Fourier transform: $\hat{A} = \text{gft}(A)$, $\hat{f} = \text{gft}(f)$.
3. For each $\rho_i \in \mathcal{R}$, solve $\hat{A}(\rho_i)\hat{x}(\rho_i) = \hat{f}(\rho_i)$.
4. Inverse transform the solution: $x = \text{igft}(\hat{x})$ and interpret the result as $\mathbf{x}$.

3 EXAMPLES

In this section, we will illustrate the method with a simple example. We choose a domain with a triangular symmetry, see Figure 3(a). As explained in Section 3.1, this domain is symmetric under a group of linear transformations known as the dihedral group of order 6, and denoted $\mathcal{D}_3$. This group is particularly illustrating, because it is the smallest non abelian group. In Section 3.2, we discuss the block diagonalization process in some detail, and in Section 3.3 we present some numerical experiments.

3.1 Triangular symmetry

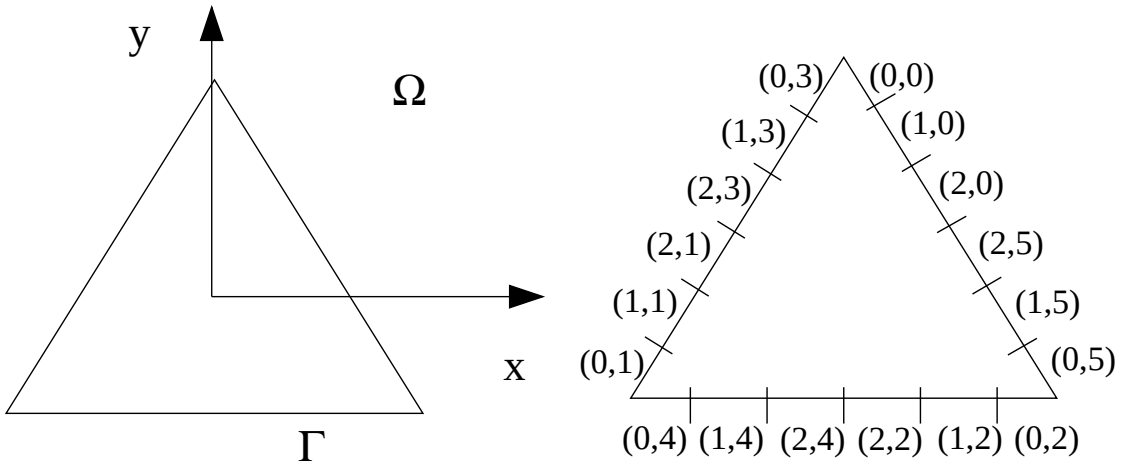
The geometry in Figure 3(a) displays a triangular symmetry. In other words, Γ is symmetric under a group of transformations, known as $\mathcal{D}_3$. This group consists of the six linear transformations which map Γ on itself. It contains the identity, rotation of 120 degrees clock-wise or counter clock-wise, and three reflections. If we denote the identity by e, clockwise rotation by a and reflection through the y-axis by b, the group may be stated

$$\mathcal{D}_3 = \langle a, b \mid a^3 = b^2 = e, bab = a^{-1} \rangle$$

and it contains the elements

$$\mathcal{D}_3 = \left\{ e, a, a^2, b, ab, a^2b \right\}.$$

We construct a discretization $\mathcal{T}_h$, that respects the symmetry, so that for any element T_j and any $g \in \mathcal{G}$, we can we can find $T_i \in \mathcal{T}_h$ such that $T_j = g(T_i)$. Furthermore, we assume



(a) (b)

Figure 3: The computational domain (a), and a symmetry respecting discretization of Γ (b). The double labeling (s, g) of the indices refer to the orbit number, s, and an enumeration of the group elements, g.

that the induced action (10) is *free*, which implies that each orbit has $|\mathcal{D}_3| = 6$ elements. Figure 3(b) illustrates how the discretization elements can be labeled using two indices, where the first index refers to the orbit.

As the second index, we use an enumeration of the group elements, in such a way that the relation

$$(s, g) = (s, e)g$$

is fulfilled. Two advantages with this labeling scheme is that the group action becomes particularly simple to compute, and it is also very simple to construct a selection $\mathcal{S}$ of orbit representatives; let for example $\mathcal{S} = \{(s, e)\}$ for all orbits $s = 0, \ldots, m - 1$.

Based on this discretization, $\mathcal{T}_h$, we construct the system matrix $\mathbf{A}$ and the right hand side f, as outlined in Section 2.1. However, since $\mathbf{A}$ is equivariant we only need to compute $m^2 \, |\mathcal{G}|$ different entries, and we use our labeling scheme to conveniently store $\mathbf{A}$ as A and $\mathbf{f}$ as f via

$$A_{s,t}(g) = \mathbf{A}_{(s,g),(t,e)}, \;\; f_s(g) = \mathbf{f}_{(s,g)}.$$

3.2 Block diagonalizing using the generalized Fourier transform

The next step is to transform A and f to $\hat{A}$ and $\hat{f}$. When the generalized Fourier transform (8) is plugged into the block version (15), we explicitly obtain

$$\hat{f}_i(\rho_r) = \sum_{g \in \mathcal{G}} f_i(g) \rho_r(g) \tag{17}$$

	e	a	a^2	b	ab	a^2b
ρ_0	1	1	1	1	1	1
ρ_1	1	1	1	-1	-1	-1
ρ_2	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} \omega & 0 \\ 0 & \omega^2 \end{pmatrix}$	$\begin{pmatrix} \omega^2 & 0 \\ 0 & \omega \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & \omega \\ \omega^2 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & \omega^2 \\ \omega & 0 \end{pmatrix}$

Table 1: A complete list $\mathcal{R}$ of irreducible representations for $\mathcal{D}_3$, where $\omega = e^{i\frac{2\pi}{3}}$.

and

$$\hat{A}_{i_1,i_2}(\rho_r) = \sum_{g\in\mathcal{G}} A_{i_1,i_2}(g)\rho_r(g) \tag{18}$$

for $i, i_1, i_2 \in \mathcal{S}$ and for each $\rho_r \in \mathcal{R}$. In the case of $\mathcal{D}_3$, the complete list $\mathcal{R}$ of irreducible representations contains 3 representations of dimensions $d_0 = d_1 = 1$ and $d_2 = 2$. Table 1 shows one possible choice of $\mathcal{R}$. When $d_r > 1$, entries of $\hat{A}$ and $\hat{f}$ can be written

$$\hat{f}_i(\rho_r)_{\ell_1,\ell_2} = \sum_{g\in\mathcal{G}} f_i(g)\rho_r(g)_{\ell_1,\ell_2}, \tag{19}$$

and

$$\hat{A}_{i_1,i_2}(\rho_r)_{\ell_1,\ell_2} = \sum_{g\in\mathcal{G}} A_{i_1,i_2}(g)\rho_r(g)_{\ell_1,\ell_2} \tag{20}$$

for $i, i_1, i_2 \in \mathcal{S}, \rho_r \in \mathcal{R}$, and for $\ell_1, \ell_2 = 0, \ldots, d_r - 1$.

For each ρ_r, we solve a linear system of equations

$$\hat{A}(\rho_r)\hat{x}(\rho_r) = \hat{f}(\rho_r), \tag{21}$$

where $\hat{A}(\rho_r)$ is $md_r \times md_r$, and where $\hat{x}(\rho_r)$ and $\hat{f}(\rho_r)$ are $md_r \times d_r$ matrices. In our example with $m = 3$ orbits, the block diagonal system $\hat{A}\hat{x} = \hat{f}$ has the structure indicated in Figure 4.

In order to obtain good efficiency, we need to consider in some detail how our data structures actually are stored. Regarding the generalized Fourier transform, it is often possible

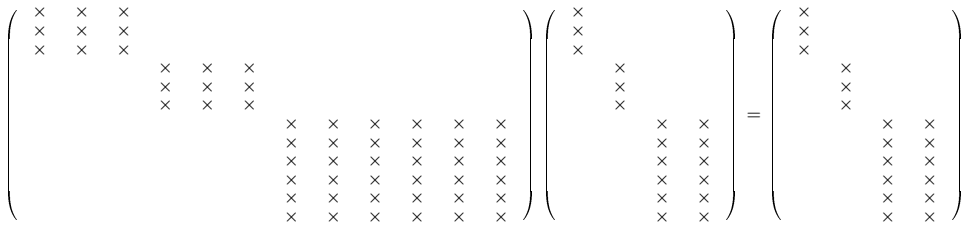


Figure 4: The transformed equation system.

to rewrite the algorithm to obtain faster transforms, see [5, 6], but since the transforms are not a bottleneck in our applications, we focus on the general algorithm (8). For the implementation of the sums to perform efficiently it is preferable that $f_i(g)$ and $A_{i_1,i_2}(g)$, as well as the $\rho_r(g)$, are stored in such a way that entries belonging to consecutive g indices are contiguous in memory. $\hat{A}$ and $\hat{f}$ can be stored either with consecutive r-indices stored contiguously, or with consecutive i- or i_1, i_2-indices stored contiguously. By storing the i-type-indices contiguously we make it easy to separate the different ρ_r-components. This is essential, because for each ρ_r, the equation system (21) is independent.

The way $\hat{A}$, $\hat{x}$ and $\hat{f}$ are stored strongly influences the performance of the numerical solution of the transformed equations. For each $\rho_r \in \mathcal{R}$ and for each i_1, ℓ_1 and ℓ_3, we explicitly solve

$$\sum_{i_2,\ell_2} \hat{A}_{i_1,i_2}(\rho_r)_{\ell_1,\ell_2} \hat{x}_{i_2}(\rho_r)_{\ell_2,\ell_3} = \hat{f}_{i_1}(\rho_r)_{\ell_1,\ell_3}.$$

If $d_r = 1$, i.e., if the irreducible representation in question is a scalar, then we only have to decide whether to store consecutive i_1 or i_2 indices contiguously, corresponding to row-wise or column-wise storage, respectively. In software matrix libraries there are efficient routines for both cases.

If $d_r > 1$ we need to construct two new composite indices based on i_1 and ℓ_1 on one hand and i_2 and ℓ_2 on the other. This could either be $n_{1a} = i_1 + m * \ell_1$ and $n_{2a} = i_2 + m * \ell_2$ (i-indices contiguous), or $n_{1b} = d_r * i_1 + \ell_1$ and $n_{2b} = d_r * i_2 + \ell_2$ (ℓ-indices contiguous). The speed of the solve is unaffected by this choice.

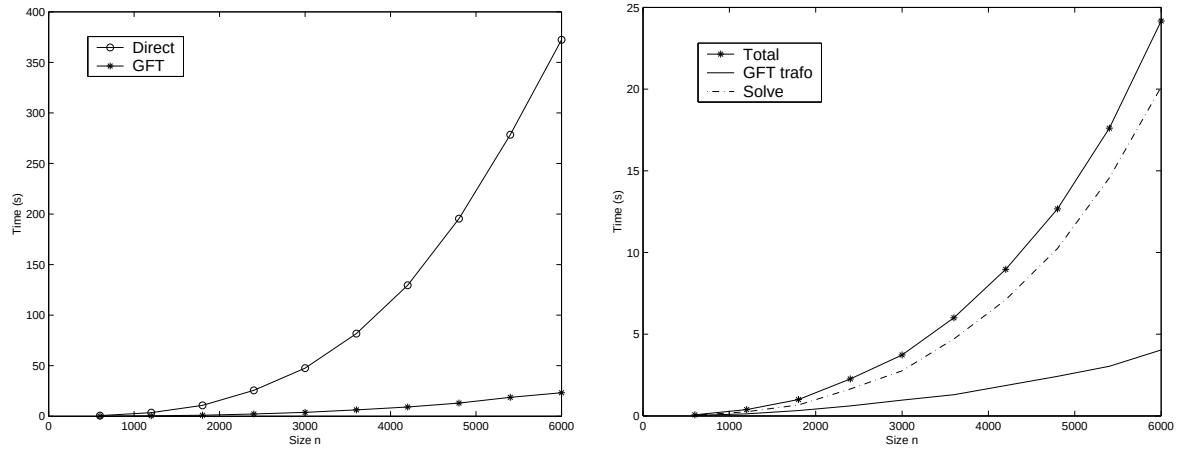
3.3 Numerical Experiments

In order to demonstrate the feasibility of the approach, we have implemented Matlab routines which solve equivariant systems of equations for a few different groups, including $\mathcal{D}_n$, (i.e., the symmetries of a regular n-sided polygon), the symmetries of a tetrahedron, and the symmetries of a cube.

In Figure 5(a), we consider the numerical solution of $\mathbf{Ax} = \mathbf{f}$, where $\mathbf{A}$ is equivariant under a free action of $\mathcal{D}_3$. We show timings where the system is solved with generalized Fourier transforms, on one hand, and with a direct solve based on LU factorization, on the other. According to theory, the LU solve is $\mathcal{O}(n^3)$ where $n = |G|\,|\mathcal{S}| = 6m$ for $G = \mathcal{D}_3$. Each transform (8) requires $\mathcal{O}(|G|^2)$ which gives in total $\mathcal{O}(|G|^2(m^2 + 2m)) = \mathcal{O}(|G|^2 m^2) = \mathcal{O}(n^2)$ for the block versions. The solution of the transformed systems is $2\mathcal{O}(m^3) + \mathcal{O}((2m)^3) \approx \mathcal{O}(\frac{1}{20}n^3)$. As the graph in Figure 5 shows, the gain in solution time is of this order.

According to theory, most of the time spent in the generalized Fourier transform based computation is on the direct solve used for finding the components of $\hat{x}(\rho_r)$. As is illustrated in Figure 5(b), the time spent on performing the gft and igft transforms is comparatively small.

The numerical experiments illustrate that for this type of problems a significant in-



(a) (b)

Figure 5: Comparison of execution times for for different problem sizes. In (a), we compare direct solve with a generalized Fourier transform (GFT) based solve. In (b), we illustrate that for the GFT based solve, the time spent on the transforms is small, compared to the time for solving the transformed system.

crease in performance can be obtained by using symmetry exploiting methods, based on the generalized Fourier transform. For equivariance under other groups, such as the symmetries of a cube, the method is even more efficient because the block diagonalization yields a larger number of independent blocks.

4 SOFTWARE ASPECTS

We have implemented the generalized Fourier transform for a few particular groups in Matlab, in order to demonstrate the feasibility of the method. Our aim is, however, to develop software for generalized Fourier transforms which is both general and efficient. We believe that generic programming is a good approach for addressing these issues, and this section is devoted to discussing a C++ design based on generic programming [7].

4.1 Groups

A mathematical group is a key abstraction in our software domain. How can we design a software abstraction for a group in such a way that other abstractions can reuse it? A first attempt may be to regard Group as a class, which supplies an interface to group operations. If the class is initialized with appropriate tables for the group operation and the inverse, any group can be represented by an object of this class. A specific element of the group may be represented simply by an integer:

```
class Group
{
```

```
public:
  Group(...); // Init appropriately
  int inverse(int g);
  int identity();
  int op(int g, int h);
  //...
};
```

This design allows us to write generic algorithms in terms of any group. Example:

```
// Returns true if g is inverse of h
bool checkInverse(Group &G, int g, int h)
{
  return G.op(g, h) == G.identity();
}
```

There are, however, two major drawbacks with this approach. First, we can not provide specialized implementations for different groups. For a cyclic group $\mathcal{C}_n$, for example, the group operation which multiplies $g = a^i$ with $h = a^j$ should utilize the formula $gh = a^{i+j \bmod n}$ instead of utilizing a table lookup. The second, and more serious, problem, is that the design is not type safe.

The first drawback may be resolved by using inheritance.

```
class Group
{
public:
  Group();

  // abstract member functions:
  virtual int op(int g, int h) = 0;
  virtual int inverse() = 0;
  virtual int identity() = 0;
  //...
};

class CyclicN : public Group
{
public:
  CyclicN(int N_) : N(N_) {}

  int op(int g, int h) {
    return (g+h) % N;
  }
  // ...
private:
```

```
  int N;
};
```

Thus, we can easily provide specialized implementations for different groups, but the second drawback remains. The design is still not type safe, and a careless user might for instance try to add elements from different groups such as apples and pears. Another disadvantage with the inheritance based approach is that the dynamic binding may decrease the performance. Generic programming offers a solution to all of these problems.

With generic programming, we implement different groups in different classes, which all adhere to an abstractly defined interface. Algorithms and data structures which are based on groups are then parameterized over the group, using a template parameter. Different elements of a specific group are now objects of the class representing this group, ensuring type safety. Example:

```
template<class G>
bool checkInverse(const G &g, const G &h)
{
  return g*h == G::identity();
}
```

Our parameterized version of `checkInverse` takes two objects of group `G`. The code will compile provided the class `G` supports a group operation (overloaded as `operator*`), an equality operator, and a static member function `identity()`. Cyclic groups for example, may be provided as follows.

```
template<int N>
class Cyclic
{
public:
  Cyclic(int i_=0) : i(i_) {}

  static Cyclic<N> identity() {
    return Cyclic<N>(0);
  }
  Cyclic<N> operator*(Cyclic<N> h) const {
    return Cyclic<N>((i+h.i) % N);
  }
  bool operator==(const Cyclic<N> &g) const {
    return i == g.i;
  }
  //...
private:
  int i;
};
```

In this way, both type safety and implementation efficiency is handled. For instance, we may not multiply objects of `Cyclic<4>` with objects of `Cyclic<5>`. Also note that the function calls are bound in link time, which avoids the run-time overhead of dynamic binding.

The group abstraction also provides facilities for iterating over all elements of the group, in order to separate data structures from algorithms. See Section 4.4 for an example.

4.2 Vector spaces

A vector space V over a field F (usually $\mathbb{C}$ or $\mathbb{R}$) supports (linearly) the following operations

$$\begin{aligned} + &: V \times V \to V \\ \cdot &: F \times V \to V \end{aligned}$$

Again, we want to use generic programming techniques to obtain efficient and type safe software. Here, we will mention a particular detail which must be addressed. In order to use the vector space abstraction in a generic fashion, it is required to publicize its underlying field. That is, we want `Field` to be a public typename for a class which implements a vector space. But this requirement can not be implemented for basic types, which also are vector spaces! For example, $\mathbb{R}$ is a vector space over itself, and we can not change the implementation of the basic data types such as `double`. This is a situation where traits [15] come to the rescue. For every vector space abstraction, we parameterize a trait `VectorSpaceTrait` which publicize the underlying field. A generic implementation of this trait relies on the vector space itself to provide the underlying field, but for instance for `double` and `complex`, we need to provide specialized implementations.

```
// Generic implementation
template<class VectorSpace>
struct VectorSpaceTrait {
  typedef typename VectorSpace::Field Field;
};

// double and complex need special implementations
template<>
struct VectorSpaceTrait<double> {
  typedef double Field;
};

template<class F>
struct VectorSpaceTrait<complex<F> > {
  typedef complex<F> Field;
};
```

Two examples of vector spaces which use this abstraction are matrix spaces such as $\mathbb{C}^{m\times n}$ and group spaces $\mathbb{C}^G$, which will be discussed next.

4.3 Matrix spaces and group spaces

We define a matrix space $V^{m\times n}$ as a vector space over the same field as V, which in addition to the vector space operations also supplies an index operator which given two indices returns an element of vector space V. We outline a crude implementation:

```
template<class V, int M, int N>
class MatrixSpace
{
public :
  typedef typename VectorSpaceTrait<V>::Field Field;

  V &operator()(int i, int j) {
    return data[i][j];
  }
  // ...
private :
  V data[M][N];
};
```

A specific matrix space is instantiated by providing appropriate template parameters. For example, $A \in \mathbb{R}^{2\times 3}$ would be declared as `MatrixSpace<double,2,3> A;`

A vector space such as $\mathbb{C}^n$ is isomorphic to $\mathbb{C}^{n\times 1}$, and we can therefore use `MatrixSpace` also in this case. It is possible to partially specialize `MatrixSpace` into a column vector space, but it requires a level of detail which is not suitable for this presentation.

Another vector space which we, however, need to discuss, is the group space $\mathbb{C}^{\mathcal{G}}$. It is used to represent entities in the group algebra $\mathbb{C}\mathcal{G}$ as discussed on page 5. The basic difference between a matrix space and a group space is the index operation: a group space is a vector space indexed with group elements.

```
template<class V, class G>
class GroupSpace
{
public:
  typedef typename VectorSpaceTrait<V>::Field Field;

  V &operator()(const G &g) {
    return data[ int(g) ];
  }
  // ...
private:
  V data[ G::SIZE ];
};
```

The definition of these vector spaces provides us with great flexibility. For instance, we may represent elements in $\mathbb{C}^{m\times m}\mathcal{G}$ either as objects of class

```
typedef MatrixSpace<GroupSpace<complex,Group>,M,M> MatrixGroupSpace;
```

or as objects of class

```
typedef GroupSpace<MatrixSpace<complex,M,M>,Group> GroupMatrixSpace;
```

4.4 Multiplication

In addition to be able to represent the various vector space abstractions, we must also provide basic operations such as multiplication between vector spaces. In order to cater for generic multiplication operators, we decompose multiplication into independent sub-algorithms. The first point to notice is that a multiplication always may be conceived as consisting of two parts. The result is first set to zero and contributions are then added to the result. To be specific, we use the following parameterized algorithm.

```
template<class U, class V, class W>
void mult(const U &u, const V &v, W &w)
{
  setZero(w);     // w = 0
  addMult(u,v,w); // w += u*v
}
```

We assume that zeroing a vector is supported by the vector space, and we focus on the `addMult` operation. For basic types, we specify overloaded instances of `addMult`, for instance

```
inline void addMult(const double &u, const double &v, double &w)
{
  w += u*v;
}
```

More interesting, however, is the implementation of matrix multiplication. Given $* : U \times V \to W$, we lift this multiplication to $* : U^{m\times n} \times V^{n\times p} \to W^{m\times p}$:

```
template<class U, class V, class W, int M, int N, int P>
void addMult(const MatrixSpace<U,M,N> &u,
             const MatrixSpace<V,N,P> &v,
             MatrixSpace<W,M,P> &w)
{
  for(int i=0; i<M; i++)
    for(int j=0; j<N; j++)
      for(int k=0; k<P; k++)
        addMult(u(i,j), v(j,k), w(i,k));
}
```

In a similar fashion as matrix multiplication was implemented, we implement the convolution operator $* : U^G \times V^G \to W^G$. This example also illustrates the means of iteration which the group `G` is required to supply.

```
template<class U, class V, class W, class G>
void addMult(const GroupSpace<U,G> &u,
             const GroupSpace<V,G> &v,
             GroupSpace<W,G> &w)
{
  for(G g = G::begin(); g != G::end(); g++)
    for(G h = G::begin(); h != G::end(); h++)
      addMult(u(g), v(h), w(g*h));
}
```

We now obtain two equivalent block convolution implementations, depending on how we represent $\mathbb{C}^{m\times n}G$. The type `MatrixGroupSpace` provides multiplication where the group convolution is performed in the inner loops, whereas `GroupMatrixSpace` executes the group multiplication in the outer loops, see equation (13) and equation (14), respectively.

5 SUMMARY

We have discussed how to use the generalized Fourier transform to block diagonalize certain types of equivariant matrices. Such matrices may arise in various applications with geometrical symmetries, for example when the boundary element method is used to solve an electrostatic problem outside a symmetric object. The method was described in detail for an object with a triangular symmetry, and the feasibility of the method was confirmed by numerical experiments.

In this paper, however, our focus has been on the applicability of generic programming for this kind of mathematical software. Even though this research is not concluded, we believe the results to be encouraging. Since generic programming supports type safety, link time polymorphism, as well as a clean separation between data structures and algorithms, it is suitable for mathematical software. The `Group` abstraction in Section 4 illustrates this nicely. Furthermore, we find that generic programming is well suited for implementing mathematical abstraction *layers*, cf. [16]. This is exemplified by our implementation of generic multiplications for various vector spaces. A few special techniques, such as traits, are also required in order to implement useful generic abstractions.

Our plug-and-play multiplication example illustrates some of the potential a design based on generic programming has. However, in order to address efficiency, we must pay at least as much attention to the low level data structures as to the algorithms, see the discussion in Section 3.1. We are therefore refining our vector space abstractions to support various data layouts. By using different *policies* [17], for different indexing schemes, the actual ordering of the numerical data can be tailored to better match the algorithms.

Acknowledgments. This research was partly supported by a grant from the Göran Gustafsson Foundation.

REFERENCES

[1] E. L. Allgower and K. Georg and R. Miranda and J. Tausch, Numerical exploitation of equivariance. *Zeitschrift fur Angewandte Mathematik und Mechanik*, **78**, 185–201, 1998.

[2] E. L. Allgower and K. Georg *Exploiting Symmetry in Numerical Solving*, in: Proceedings of the Seventh Workshop on Differential Equations and its Applications, 1999, C.-S. Chien, editor, Taichung, Taiwan.

[3] K. Åhlander and H. Munthe-Kaas. On Applications of the Generalized Fourier Transform in Numerical Linear Algebra, *Document in preparation*

[4] M. Bonnet, Exploiting partial or complete geometrical symmetry in 3D symmetric Galerkin indirect BEM formulations. *Intl. J. Numer. Meth. Engng.*, **57**, 1053–1083, 2003.

[5] D. Maslen and D. Rockmore, Generalized FFTS - A Survey of Some Recent Results, *Technical Report PCS-TR96-281, Dartmouth College, Computer Science, Hanover, NH*, 1996.

[6] D. Rockmore, Some applications of generalized FFTs, *Proceedings of the DIMACS Workshop on Groups and Computation*, June 7-10, 1995, eds. L. Finkelstein and W. Kantor, (1997) 329–369

[7] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley, 2000, ISBN 0-201-30977-7

[8] D. Musser and G. Derge and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 2001, ISBN: 0201379236

[9] C. Johnson. *Numerical solution of partial differential equations by the finite element method.* Studentlitteratur, 1987, ISBN 91-44-25241-2

[10] C. Johnson and R. Scott, An Analysis of Quadrature Errors in Second-Kind Boundary Integral Methods, *SIAM J, on Num. Anal*, **26(6)**, 1356–1382, 1989.

[11] P.G. Martinsson and V. Rokhlin, A fast direct solver for boundary integral equations in two dimensions, *Yale research report YALEU/DCS/RR-1264.*

[12] J. P. Serre. *Linear Representations of Finite Groups.* Springer, 1977, ISBN 0387901906

[13] G. James and M. Liebeck. *Representations and Characters of Groups, 2nd Edition.* Cambridge University Press, 2001, ISBN: 052100392X.

[14] J. S. Lomont. *Applications of Finite Groups*. Academic Press, New York, 1959.

[15] N. Myers, Traits: A New and Useful Template Technique *C++ Report.*, **7**, June, 1995.

[16] Åhlander, K. and Haveraaen, M. and Munthe-Kaas, H., On the Role of Mathematical Abstractions for Scientific Computing, in *The Architecture of Scientific Software*, Editors R. Boisvert and P. Tang, Kluwer Academic Publishers, Boston, 145–158, 2001.

[17] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001, ISBN: 0201704315.