1 Introduction

DIRECTFN package is free software for the accurate and efficient evaluation of four-dimensional singular integrals arising in Galerkin Method of Moments surface integral equation formulations over conforming triangular or quadrilateral meshes. The fully-numerical algorithms of DIRECTFN [1] are suitable for the following applications:

- Weakly and strongly singular kernels
- Planar and curvilinear elements
- Basis/testing functions of arbitrary order
- Problem-specific Green's functions (e.g. expressed in spectral integral form)
- From zero frequency to frequencies beyond microwaves
- Spectral convergence to machine precision

In general case, DIRECTFN library is intended to compute the following weakly (I_{WS}) and strongly (I_{SS}) singular integrals:

$$I_{WS} = \int\limits_{E_P} \int\limits_{E_Q} G(\mathbf{r}, \mathbf{r}') dS' dS, \tag{1}$$

$$I_{WS}^{m,n} = \int\limits_{E_P} \mathbf{g}_m(\mathbf{r}) \cdot \int\limits_{E_Q} G(\mathbf{r}, \mathbf{r}') \cdot \mathbf{f}'_n(\mathbf{r}') dS' dS, \tag{2}$$

$$I_{SS}^{m,n} = \int\limits_{E_P} \mathbf{g}_m(\mathbf{r}) \cdot \int\limits_{E_Q} (\nabla G(\mathbf{r}, \mathbf{r}') \times \mathbf{f}'_n(\mathbf{r}')) dS' dS. \tag{3}$$

Here E_P and E_Q are observation and source elements, i.e., two triangles or two quadrilaterals, which may coincide (self-term case), have a common edge (edge adjacent case) or a common vertex (vertex adjacent case); $\mathbf{f}_n(\mathbf{r}')$ are basis and $\mathbf{g}_m(\mathbf{r})$ are testing functions; $G(\mathbf{r}, \mathbf{r}')$ by default is homogeneous Green's function: $G(\mathbf{r}, \mathbf{r}') = \frac{e^{-ik|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}$. However, it can be replaced by any function of source $\mathbf{r}'$ and observation $\mathbf{r}$ points $K(\mathbf{r}, \mathbf{r}') \sim |\mathbf{r} - \mathbf{r}'|^{-p}, \quad p = 1, 2$.

There are several different sets of basis and testing functions implemented in DIRECTFN, and they will be described later in details.

2 Download

The source code of DIRECTFN is written in C++11 and can be downloaded from Github repository [2]. Also, mex plugins are available to provide a fast Matlab interface to this functionality. The repository is structured as follows:

- `./include` folder contains headers of the library with declarations only.
- `./src` folder contains source files where all the templates are instantiated in corresponding files properly. So, in the client programs one can use the templated classes which have been precompiled already.
- `./lib` is the target path to store the static library, which can be further used, e.g. in Matlab interface, or python wrappers.
- `./tests` directory contains some initial examples of how to use the library in your own code.
- `./examples` folder contains the scripts to reproduce results, presented in papers [3] and [4].

3 Compilation

The provided static library and examples can be compiled using make utility. Therefore, it requires Windows users to have some Linux-like environment, such as Cygwin or MinGW. However,the other possible option is to use nmake utility from Visual Studio (see 3) . For the compilation you should do the following steps:

- First, go to the `./settings` directory and create or copy-modify platform-specific file `Makefile.your_cpu_alias`. Here the `your_cpu_alias` is an alias for your system compiler (like intel, gcc620, clang or so). This alias is just an abbreviation to distinguish different Makefiles and can be any name you like.
- Next, define environment variable `CPU=your_cpu_alias`. It can be done by the command

  ```
  $ export CPU=your_cpu_alias
  ```

  or by figuring out the CPU explicitly when compiling the code with make utility:

  ```
  make CPU=your_cpu_alias
  ```

- Add the following lines to the end of `./settings/Makefile.in`:

  ```
  ifeq ($(CPU),your_cpu_alias)
  include $(MF_PREFIX)/Makefile.target
  endif
  ```

- Finally, go to the `./lib/unix` folder and type

  ```
  $ make
  ```

  or

  ```
  $ make CPU=your_cpu_alias
  ```

  This command will compile the code and move the static library into the `./lib` folder. Default name for the library is `directfn` (thus, it will be compiled and builded in the `libdirectfn.a` after compilation). You can change the name in `settings/Makefile.in` by defining `DIRECTFN_LIB_NAME` variable.

Once the library is builded, you may go to the `./tests` or `./examples` folders to compile and run some examples.

Compilation with nmake from Visual Studio

In order to able working with the nmake utility, we provide also special Makefiles for it, since the syntax is slightly different from the one that make uses. To compile the source code with nmake you should do the following simple steps:

- Run the Developer Command Prompt for Visual Studio and change the folder to where you have cloned the repository.
- Go to the folder `./lib/win/` and build the static library by typing

  ```
  nmake
  ```

- To compile the examples run the Developer Command Prompt for Visual Studio or Visual Studio x86(x64) Native Tools, change the folder to `.examples/<Paper Name>/c++` and type

  ```
  nmake -f Makefile_win
  ```

4 Examples

The folder `./examples` in the repository contains the scripts to reproduce results, presented in [3] and [4]. For a given example there are two ways to execute it:

- Go to `./examples/<Paper name>/c++/` folder and compile the C++ code by

  `make`

  or

  `nmake -f Makefile_win`

  if you use Windows utility `nmake`. then type the name of the executable file

  `./test_name`

  or just

  `test_name`

  while using Visual Studio command prompt. It will create a file `Results_<test_name>.txt` with timings and errors, and you can open `Examples.ipynb` notebook and visualize the results by executing the corresponding cell. Note that to run the .ipynb file you need ipython to be installed.

- The other way is to use Matlab scripts. To do this, you first need to build the mex interface to C++ functions by going to the `./mex` folder and executing the `build.m` script. After that, you can switch to the folder `./examples/EuCap2017_Examples/matlab` or `./examples/Full_Paper_Examples/matlab` and run any `test_name.m` script and it will do all the computations and produce the error plots and timings automatically.

- Note that if you're going to build mex files with one of the standart Windows compiler, e.g., provided by Visual Studio, you should build the static library with Windows compiler as well, i.e., using nmake.

5 Library Usage

5.1 Basic algorithm

The complete manual of how to use DIRECTFN library can be found in `./docs/Manual`. Here we briefly describe the main steps of computing the surface-surface singular integrals. In C++, you first

```
#include "directfn_quad.h"
```

if you need to work with quadrilateral elements, or

```
#include "directfn_triag.h"
```

if you need to work with triangles. Then you should define a "singular contour" `SingularContour3xn` according to adjacency type with the proper number of points:

```
SingularContour3xn cntr;
cntr.set_points(r1, r2, r3, r4);
```

Here `double[3] ri` are coordinates of vertices of the considered elements in 3D–space. Note that the order of vertices for each case of adjacency must correspond to the rules defined below in Sect. 6 for quadrilaterals and Sect. 7 for triangles.
Next, create an object of `Quadrilateral_ST(EA,VA)` algorithm () and use its interface instantiated by a `ParticularKernel` kernel type.

```
unique_ptr<Quadrilateral_ST<ParticularKernel>>
        up_quad_st(new Quadrilateral_ST<ParticularKernel>());
```

The type `ParticularKernel` is a C++ type which inherits the `AbstractKernel` class defined in `directfn_kernel_base.h/cpp`. Here and next we denote as "Kernel" all specific kernels implemented for quadrilaterals and triangles are discussed in details in the next sections.
Now you should define the wavenumber `k0` and the orders of Gauss-Legendre quadratures `N1, N2, N3, N4`, and assign your contour `cntr` with vertices to your object `up_quad_st`.

```
up_quad_st.set_wavenumber(k0);
up_quad_st.set_Gaussian_orders_4(N1, N2, N3, N4);
up_quad_st.set(cntr);
```

Now you are ready to calculate the surface-surface singular integral(s) by calling

```
up_quad_st.calc_Iss(); // Iss = Integral Surface-Surface
```

and use the obtained values.

```
const dcomplex * ref_val = up_quad_st->Iss();
// use ref_val somewhere
```

If you need to recalculate these integrals for several contours or wave numbers we strongly recommend to do it as it is shown above: create an object of the `Quadrilateral_ST(EA,VA)` algorithm once and then use it for contour points or other parameters reset. This helps to avoid time-consuming memory allocation/deallocation related to creating and destroying the object.

In case you need to compute the integral once, you can use more simple interface mostly developed for usage inside the Matlab scripts. (See Sect. 8 and file `directfn_quad.h`).

6 Quadrilateral elements

This section contains instructions how to use DIRECTFN to compute singular integrals over quadrilaterals. All the examples can be found in the repository in folder `./tests/basic`. The corresponding source files are `test_directfn_quad_st.cpp`, `test_directfn_quad_ea.cpp` and `test_directfn_quad_va.cpp` for self-term(ST), edge adjacent (EA) and vertex adjacent (VA) cases, respectively.

6.1 Setting the points

As it were mentioned in 5.1, we first need to set the points of a "singular contour" `SingularContour3xn`. There are two types of quadrilateral elements, implemented in DIRECTFN: the four-node flat or bilinear quadrilaterals and nine-node quadratic quadrilaterals [5]. Therefore, for the first type you need to specify:

- 4 points for self-term case:

  ```
  cntrST.set_points(r1, r2, r3, r4);
  ```

- 6 points for edge-adjacent case:

  ```
  cntrEA.set_points(r1, r2, r3, r4, r5, r6);
  ```

- 7 points for vertex-adjacent case:

```
cntrVA.set_points(r1, r2, r3, r4, r5, r6, r7);
```

In case of using curvilinear elements, you should specify:

- 9 points for self-term case:

```
cntrST.set_points(r1, r2, r3, r4, r5, r6, r7, r8, r9);
```

- 15 points for edge-adjacent case:

```
cntrEA.set_points(r1, r2, r3, r4, r5, r6, r7, r8, r9, r10,
        r11, r12, r13, r14, r15);
```

- 17 points for vertex-adjacent case:

```
cntrVA.set_points(r1, r2, r3, r4, r5, r6, r7, r8, r9, r10,
        r11, r12, r13, r14, r15, r16, r17);
```

Note that the nodes of the associated quadrilaterals must follow the orientation depicted in Fig.1 for 4-node and Fig.2 for 9-node quadrilaterals.

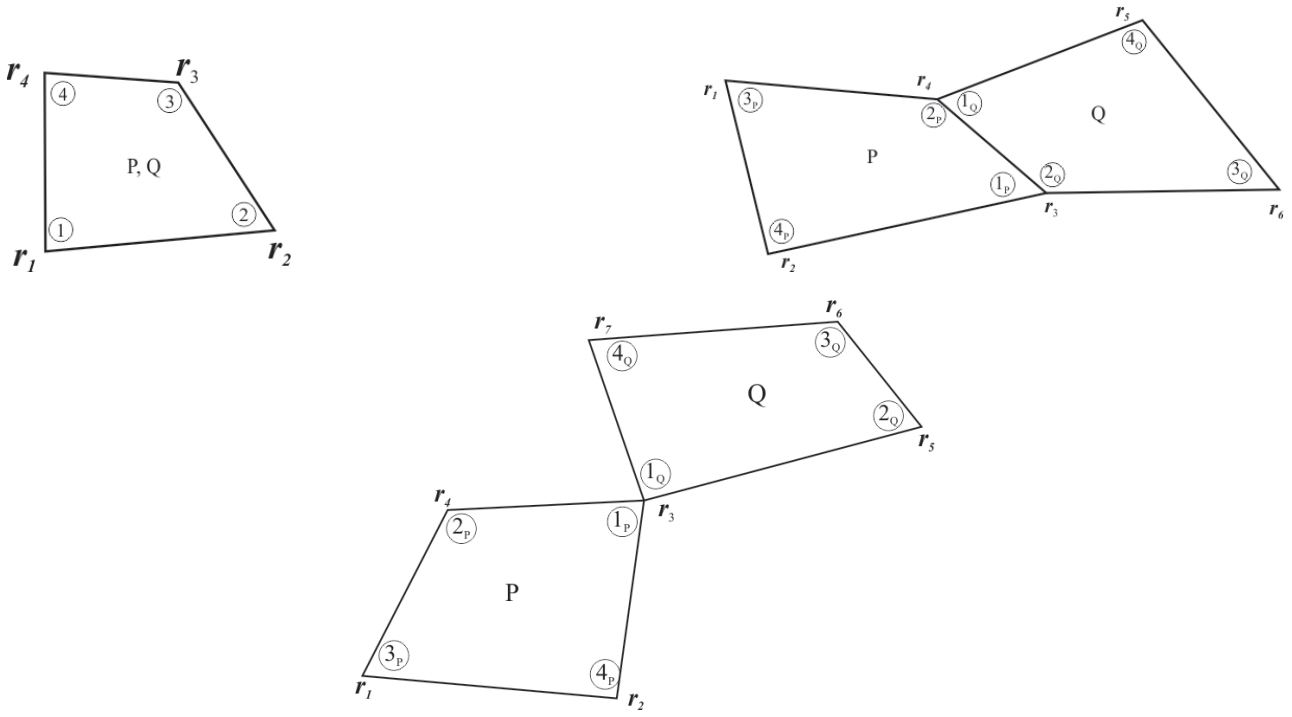

Figure 1: Orientation of 4-node quadrilateral elements in DIRECTFN: (a) self-term case (b) edge adjacent case; (c) vertex adjacent case.

6.2 Choosing the basis and testing functions.

Constant case

The first and the simplest case implemented is the weakly singular integral with constant basis and testing functions:

$$I_{WS} = \int\limits_{E_P} \int\limits_{E_Q} G(\mathbf{r}, \mathbf{r}') dS' dS, \tag{4}$$

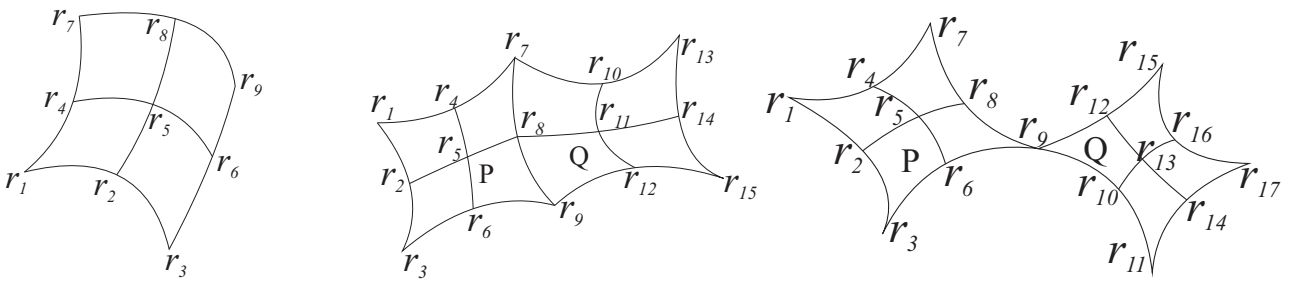

Figure 2: Orientation of curvilinear quadrilateral elements in DIRECTFN:(a) self-term case; (b) edge adjacent case; (c) vertex adjacent case.

To compute this integral, after setting the contour, you should create an object of the ST,EA or VA algorithm `Quadrilateral_ST(EA,VA)` with the type of the kernel `<QuadrilateralKernel_PlanarScalar>`, if you work with planar elements

```
unique_ptr<Quadrilateral_ST<QuadrilateralKernel_PlanarScalar>>
        up_quad_st(new Quadrilateral_ST<QuadrilateralKernel_PlanarScalar>());
```

or with the kernel of the type `<QuadrilateralKernel_CurvilinearScalar>`, if you use curvilinear elements:

```
unique_ptr<Quadrilateral_ST<QuadrilateralKernel_CurvilinearScalar>>
        up_quad_st(new Quadrilateral_ST<QuadrilateralKernel_CurvilinearScalar>());
```

Vector Basis and Testing Functions

The other option is to compute the following weakly and strongly singular integrals:

$$I_{m,n}^{WS} = \int\limits_{E_P} \mathbf{f}_m(\mathbf{r}) \cdot \int\limits_{E_Q} G(\mathbf{r},\mathbf{r}') \cdot \mathbf{f}'_n(\mathbf{r}')dS'dS, \tag{5}$$

$$I_{m,n}^{SS} = \int\limits_{E_P} \mathbf{f}_m(\mathbf{r}) \cdot \int\limits_{E_Q} (\nabla G(\mathbf{r},\mathbf{r}') \times \mathbf{f}'_n(\mathbf{r}'))dS'dS, \tag{6}$$

where E_P and E_Q are observation and source quadrilateral elements respectively, and $G(\mathbf{r},\mathbf{r}') = \frac{e^{-ik|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}$ is homogeneous Green's function. Here $\mathbf{f}_m(\mathbf{r})$ and $\mathbf{f}_n(\mathbf{r}'), (m,n = 1,2,3,4)$ are vector basis functions of the 1st order [6, 7]. For a given quadrilateral the four vector functions, associated with its edges are

$$\begin{aligned} \mathbf{b}_1 &= (v-1)\frac{\mathbf{r}_v}{J(u,v)}, \quad \mathbf{b}_2 = (u+1)\frac{\mathbf{r}_u}{J(u,v)}, \\ \mathbf{b}_3 &= (v+1)\frac{\mathbf{r}_v}{J(u,v)}, \quad \mathbf{b}_4 = (u-1)\frac{\mathbf{r}_u}{J(u,v)}. \end{aligned}, \tag{7}$$

where

$$J(u,v) = |\mathbf{r}_u \times \mathbf{r}_v|, \tag{8}$$

and

$$\mathbf{r}_u \equiv \frac{\partial \mathbf{r}}{\partial u} = \frac{-\mathbf{r}_1 + \mathbf{r}_2 + \mathbf{r}_3 - \mathbf{r}_4 + v\,(\mathbf{r}_1 - \mathbf{r}_2 + \mathbf{r}_3 - \mathbf{r}_4)}{4}, \tag{9}$$

$$\mathbf{r}_v \equiv \frac{\partial \mathbf{r}}{\partial v} = \frac{-\mathbf{r}_1 - \mathbf{r}_2 + \mathbf{r}_3 + \mathbf{r}_4 + u\,(\mathbf{r}_1 - \mathbf{r}_2 + \mathbf{r}_3 - \mathbf{r}_4)}{4}. \tag{10}$$

To compute these integrals over planar elements, you should create an object of the ST,EA or VA algorithm `Quadrilateral_ST(EA,VA)` with the type of the kernel `<QuadrilateralKernel_PlanarVectorWS>` for weakly singular integrals(5)

```
unique_ptr<Quadrilateral_VA<QuadrilateralKernel_PlanarVectorWS>>
       up_quad_va(new Quadrilateral_VA<QuadrilateralKernel_PlanarVectorWS>());
// WS = Weakly Singular
```

and with the kernel of the type `<QuadrilateralKernel_PlanarVectorSS>` for strongly singular integrals(6):

```
unique_ptr<Quadrilateral_EA<QuadrilateralKernel_PlanarVectorSS>>
        up_quad_ea(new Quadrilateral_EA<QuadrilateralKernel_PlanarVectorSS>());
// SS = Strongly Singular
```

For curvilinear elements the syntax is almost the same,

```
unique_ptr<Quadrilateral_ST<QuadrilateralKernel_CurvilinearVectorWS>>
        up_quad_st(new Quadrilateral_ST<QuadrilateralKernel_CurvilinearVectorWS>());
// WS = Weakly Singular
```

for weakly singular integrals (5) and

```
unique_ptr<Quadrilateral_EA<QuadrilateralKernel_CurvilinearVectorSS>>
        up_quad_ea(new Quadrilateral_EA<QuadrilateralKernel_CurvilinearVectorSS>());
// SS = Strongly Singular
```

for strongly singular integrals (6). The output parameters for both cases are:

$$
\begin{aligned}
I(1) &\to I^{1,1}_{WS,SS} & I(9) &\to I^{3,1}_{WS,SS} \\
I(2) &\to I^{1,2}_{WS,SS} & I(10) &\to I^{3,2}_{WS,SS} \\
I(3) &\to I^{1,3}_{WS,SS} & I(11) &\to I^{3,3}_{WS,SS} \\
I(4) &\to I^{1,4}_{WS,SS} & I(12) &\to I^{3,4}_{WS,SS} \\
I(5) &\to I^{2,1}_{WS,SS} & I(13) &\to I^{4,1}_{WS,SS} \\
I(6) &\to I^{2,2}_{WS,SS} & I(14) &\to I^{4,2}_{WS,SS} \\
I(7) &\to I^{2,3}_{WS,SS} & I(15) &\to I^{4,3}_{WS,SS} \\
I(8) &\to I^{2,4}_{WS,SS} & I(16) &\to I^{4,4}_{WS,SS}
\end{aligned}
\tag{11}
$$

7 Triangular elements

This section contains instructions how to use DIRECTFN to compute singular integrals over triangular elements. All the examples can be found in the repository in folder `./tests/basic`. The corresponding source files are `test_directfn_triag_st.cpp`, `test_directfn_triag_ea.cpp` and `test_directfn_triag_va.cpp` for self-term(ST), edge adjacent (EA) and vertex adjacent (VA) cases, respectively.

7.1 Setting the points

As for quadrilaterals, we first need to set the points of a "singular contour" `SingularContour3xn`. For the planar triangles, implemented in DIRECTFN, you should specify:

- 3 points for self-term case:

  ```
  cntrST.set_points(r1, r2, r3);
  ```

- 4 points for edge-adjacent case:

  ```
  cntrEA.set_points(r1, r2, r3, r4);
  ```

- 5 points for vertex-adjacent case:

```
cntrVA.set_points(r1, r2, r3, r4, r5);
```

Note that the nodes of the associated triangles must follow the orientation depicted in Figs 3(a) and 3(b).

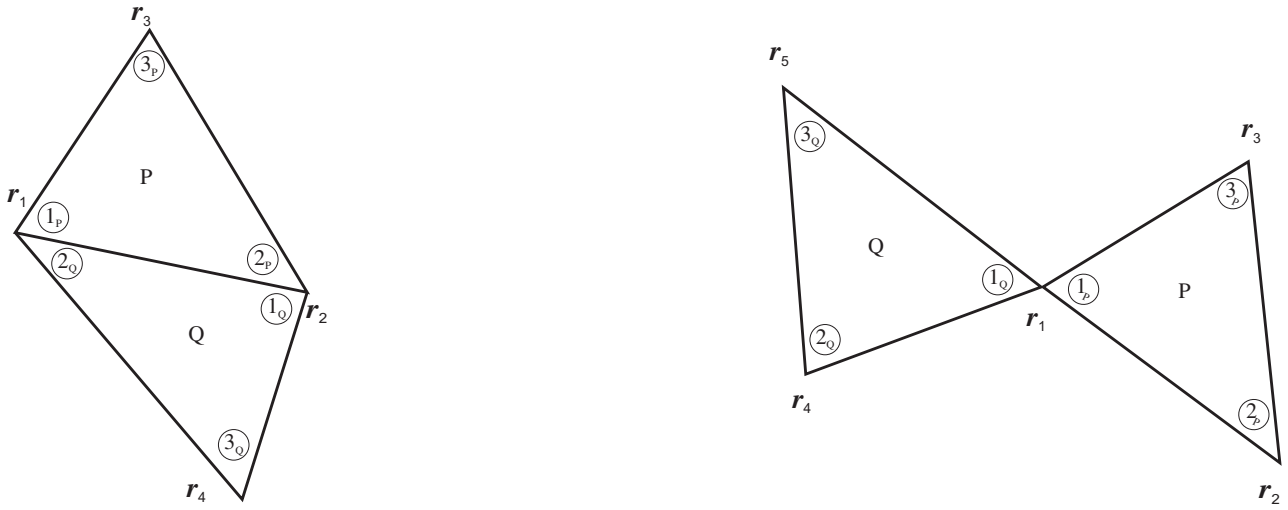

Figure 3: Orientation of triangular elements in DIRECTFN: (a) edge adjacent case; (b) vertex adjacent case.

7.2 Choosing the basis and testing functions.

Constant case

To compute the weakly singular integral with constant basis and testing functions

$$I_{WS} = \int\limits_{E_P} \int\limits_{E_Q} G(\mathbf{r}, \mathbf{r}') dS' dS, \tag{12}$$

over trinagular elements, after setting the contour, you should create an object of the ST,EA or VA algorithm `Triangular_ST(EA,VA)` with the type of the kernel `<TriangularKernel_Constant_ST(EA,VA)>`

```
unique_ptr<Triangular_ST<TriangularKernel_Constant_ST>>
        up_tri_st(new Triangular_ST<TriangularKernel_Constant_ST>());
```

Rao-Wilton-Glisson functions

The other possible choice is to compute the following weakly and strongly singular integrals

$$I^{WS}_{m,n} = \int\limits_{E_P} \mathbf{f}_m(\mathbf{r}) \cdot \int\limits_{E_Q} G(\mathbf{r}, \mathbf{r}') \cdot \mathbf{f}'_n(\mathbf{r}') dS' dS, \tag{13}$$

$$I^{SS}_{m,n} = \int\limits_{E_P} \mathbf{f}_m(\mathbf{r}) \cdot \int\limits_{E_Q} (\nabla G(\mathbf{r}, \mathbf{r}') \times \mathbf{f}'_n(\mathbf{r}')) dS' dS, \tag{14}$$

and

$$I^{SS}_{m,n} = \int\limits_{E_P} (\mathbf{n}(\mathbf{r}) \times \mathbf{f}_m(\mathbf{r})) \cdot \int\limits_{E_Q} (\nabla G(\mathbf{r}, \mathbf{r}') \times \mathbf{f}'_n(\mathbf{r}')) dS' dS, \tag{15}$$

where E_P and E_Q are observation and source flat triangular elements respectively, and $G(\mathbf{r},\mathbf{r}') = \frac{e^{-ik|\mathbf{r}-\mathbf{r}'|}}{4\pi|\mathbf{r}-\mathbf{r}'|}$ is homogeneous Green's function. To compute these integrals, you should create an object of the ST, EA or VA algorithm `TriangularKernel_ST(EA,VA)` with the types of the kernel:

- `<TriangularKernel_RWG_WS>` for weakly singular integral (13)

```
unique_ptr<Triangular_ST<TriangularKernel_RWG_WS>>
        up_tri_st(new Triangular_ST<TriangularKernel_RWG_WS>());
```

- `<TriangularKernel_RWG_SS>` for strongly singular integral (14)

```
unique_ptr<Triangular_EA<TriangularKernel_RWG_SS>>
        up_tri_ea(new Triangular_EA<TriangularKernel_RWG_SS>());
```

and

- `<TriangularKernel_nxRWG_SS>` for strongly singular integral (15)

```
unique_ptr<Triangular_VA<TriangularKernel_nxRWG_WS>>
        up_tri_va(new Triangular_VA<TriangularKernel_nxRWG_SS>());
```

The output parameters for all the cases are:

$$\begin{array}{lll} I(1)\to I^{1,1}_{WS,SS} & I(4)\to I^{2,1}_{WS,SS} & I(7)\to I^{3,1}_{WS,SS} \\ I(2)\to I^{1,2}_{WS,SS} & I(5)\to I^{2,2}_{WS,SS} & I(8)\to I^{3,2}_{WS,SS} \\ I(3)\to I^{1,3}_{WS,SS} & I(6)\to I^{2,3}_{WS,SS} & I(9)\to I^{3,3}_{WS,SS} \end{array} \tag{16}$$

8 Matlab Interface

We also provide a fast Matlab interface to have an easy access to DIRECTFN functionality. The basic Matlab syntax to call the DIRECTFN procedures is

```
I = directfn_{quad,tri}_{st,ea,va}_{plan,curv}(r1,r2,...,N1,N2,N3,N4,ko,'BasisTestingType')
```

Here the first several parameters `r1,r2,...` are positions of vertices. The required number and order of vertices is the same as in C++ and was considered earlier. `N1,N2,N3,N4` are orders of Gaussian quadratures used in four 1-D integrals, `ko` is wavenumber. The last parameter sets the type of basis and testing functions used. The possible values are:

- `'Constant'` - for both quadrilaterals and triangles, standing for integrals (4) and (12),
- `'Vector_WS'` and `'Vector_SS'` - for quadrilaterals, computing the integrals (5) and (6), and
- `'RWG_WS'`, `'RWG_SS'` and `'nxRWG_SS'` for triangles, calculating the integrals (13),(14) and (15).

9 Custom Kernels Implementation

One remarkable property of our code design is that integration algorithms are separated from the implementation of kernels to be integrated. This means that anyone who wants to use his own particular kernel should implement only the code related to the kernel and use the integration algorithms from the library by means of templates. In this case he should also instantiate the templates by the kernel type like

```
template class Quadrilateral_ST<QuadrilateralKernel_PlanarScalar>;
```

or so. Search for the line "template class Quadrilateral" in all source files to find all points of the instantiation of those types for your kernel.

If you need a kernel which depends on some parameters, you can call `kernel_ptr()` of the integration algorithm class and then change the parameters of the kernel via its own interface (all data structures including kernels are allocated inside the algorithms and uses smart pointers to avoid memory leaks).

```
ParticularKernel * const pkernel = quad_st.kernel_ptr();
for (double a = 0; a < 1.0; a += 0.1) {
    pkernel->reset_some_important_property(a);
    // computes integral with a changed
    quad_st.calc_Iss();   // Iss = Integral Surface-Surface
    const dcomplex * ref_val = up_quad_st->Iss();
    // use ref_val somewhere
}
```

There are some other properties of algorithm classes. For details, please see `directfn_interface.h/cpp` files.

The source code of DIRECFN library is structured as follows:

- `directfn_interface.h/cpp` is an abstract interface class. It provides parameters setup (number of points, singular contour, etc) and obtaining the results of computation.
- `directfn_algorithm_st.h/cpp`, `directfn_algorithm_ea.h/cpp`, and `directfn_algorithm_va.h/cpp` files contain integration algorithms for self-term, edge-adjacent and vertex-adjacent cases correspondingly for triangular and quadrilateral patches.
- `directfn_common.h/cpp` file contains some common routines, such as implementation of vector cross-product operation, or calculating normals. It includes auxiliary functions used throughout the *algorithms* implementation (like computation of limits of integration, pointers to functions and all necessary common routines).
- `directfn_gl_1d.cpp` provides data for Gauss-Legendre integration points and weights.
- `directfn_arrsum.h/cpp` implements the classes responsible for vector summation of internal subintegrals in *algorithms*. They are especially useful for cases (11) and (16).
- `directfn_contour.h/cpp`. Here the `SingularContour3xn` is implemented. It provides the contour points setup and common interface to pass them inside classes of integration *algorithms*.
- `directfn_kernel_base.h/cpp` Contains base class for *kernels* classes. They are used in the integration *algorithms*. Basically, the methods

  ```
  virtual void update_rp(const double uvxi_p[3]) noexcept = 0;
  virtual void update_rq(const double uvxi_q[3]) noexcept = 0;
  ```

  are used to setup $\mathbf{r_p}$, $\mathbf{r_q}$ values and the method

  ```
  dcomplex value(const size_t index = 0) const noexcept;
  ```

  to compute it. The input argument `index` here exactly corresponds to the equations (11) and (16).
- `directfn_kernel_quad_geom.h/cpp` contains classes responsible for planar and curvilinear geometry of quadrilateral elements.

- `directfn_kernel_quad_scal.h/cpp` contains kernels with constant basis/testing functions for quadrilaterals.
- `directfn_kernel_quad_vect.h/cpp` defines kernels with vector basis/testing functions for quadrilaterals.
- `directfn_kernel_tri.h/cpp` defines kernels related to triangular elements (constant case, RWG basis functions).
- `directfn_greenfunc.h/cpp` implements the classes (base and derived ones) related to the Green's functions, which are used inside the *kernel* classes (as a pointer to a base virtual class).

References

[1] A. G. Polimeridis, F. Vipiana, J. R. Mosig, and D. R. Wilton, "DIRECTFN: Fully numerical algorithms for high precision computation of singular integrals in Galerkin SIE methods," *IEEE Trans. Antennas Propag.*, vol. 61, no. 6, pp. 3112–3122, Jun. 2013.

[2] "DIRECTFN package," 2017. [Online]. Available: https://github.com/thanospol/DIRECTFN

[3] A. A. Tambova, G. D. Guryev, and A. G. Polimeridis, "On the fully numerical evaluation of singular integrals over coincident quadrilateral patches," *11th European Conference on Antennas and Propagation, March 2017, Paris, France.*

[4] A. A. Tambova, M. S. Litsarev, G. D. Guryev, and A. G. Polimeridis, "on the generalization of directfn for singular integrals over quadrilateral patches," *IEEE Trans. Antennas Propag. (submitted).*

[5] B. M. Kolundzija and A. R. Djordjevic, *Electromagnetic modeling of composite metallic and dielectric structures.* Artech House, 2002.

[6] J. Jin, *The Finite Element Method in Electromagnetics*, 3rd ed. Wiley-IEEE Press, 2014.

[7] M. Djordjević and B. M. Notaroš, "Double higher order method of moments for surface integral equation modeling of metallic and dielectric antennas and scatterers," *IEEE Trans. Antennas Propag.*, vol. 52, no. 8, pp. 2118–2129, Aug. 2004.