CENTER FOR EFFICIENT EXASCALE DISCRETIZATIONS

libCEED User Manual

Release 0.12.0

Ahmad Abdelfattah Valeria Barra Natalie Beams
Jed Brown Jean-Sylvain Camier Veselin Dobrev
Yohann Dudouit Leila Ghaffari Sebastian Grimberg
Tzanio Kolev David Medina Will Pazner
Thilina Ratnayaka Rezgar Shakeri
Jeremy L. Thompson Stan Tomov James Wright III

Nov 01, 2023

Contents

1 Introduction

Historically, conventional high-order finite element methods were rarely used for industrial problems because the Jacobian rapidly loses sparsity as the order is increased, leading to unaffordable solve times and memory requirements [Brown10]. This effect typically limited the order of accuracy to at most quadratic, especially because quadratic finite element formulations are computationally advantageous in terms of floating point operations (FLOPS) per degree of freedom (DOF)—see Fig. 1.1—, despite the fast convergence and favorable stability properties offered by higher order discretizations. Nowadays, high-order numerical methods, such as the spectral element method (SEM)—a special case of nodal p-Finite Element Method (FEM) which can reuse the interpolation nodes for quadrature—are employed, especially with (nearly) affine elements, because linear constant coefficient problems can be very efficiently solved using the fast diagonalization method combined with a multilevel coarse solve. In Fig. 1.1 we analyze and compare the theoretical costs, of different configurations: assembling the sparse matrix representing the action of the operator (labeled as *assembled*), non assembling the matrix and storing only the metric terms needed as an operator setup-phase (labeled as *tensor-qstore*) and non assembling the matrix and computing the metric terms on the fly and storing a compact representation of the linearization at quadrature points (labeled as *tensor*). In the right panel, we show the cost in terms of FLOPS/DOF. This metric for computational efficiency made sense historically, when the performance was mostly limited by processors' clockspeed. A more relevant performance plot for current state-of-the-art high-performance machines (for which the bottleneck of performance is mostly in the memory bandwith) is shown in the left panel of Fig. 1.1, where the memory bandwith is measured in terms of bytes/DOF. We can see that high-order methods, implemented properly with only partial assembly, require optimal amount of memory transfers (with respect to the polynomial order) and near-optimal FLOPs for operator evaluation. Thus, high-order methods in matrix-free representation not only possess favorable properties, such as higher accuracy and faster convergence to solution, but also manifest an efficiency gain compared to their corresponding assembled representations.

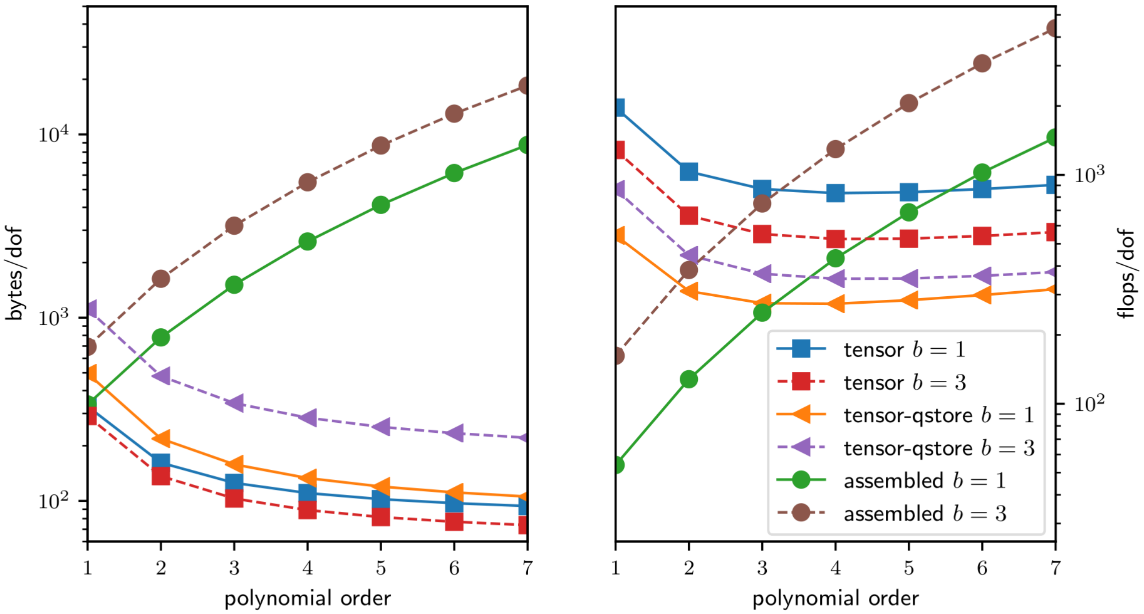


Fig. 1.1: Comparison of memory transfer and floating point operations per degree of freedom for different representations of a linear operator for a PDE in 3D with b components and variable coefficients arising due to Newton linearization of a material nonlinearity. The representation labeled as *tensor* computes metric terms on the fly and stores a compact representation of the linearization at quadrature points. The representation labeled as *tensor-qstore* pulls the metric terms into the stored representation. The *assembled* representation uses a (block) CSR format.

Furthermore, software packages that provide high-performance implementations have often been special-purpose and intrusive. libCEED [BAB+21] is a new library that offers a purely algebraic interface for matrix-free operator representation and supports run-time selection of implementations tuned for a variety of computational device types, including CPUs and GPUs. libCEED's purely algebraic interface can unobtrusively be integrated in new and legacy software to provide performance portable interfaces. While libCEED's focus is on high-order finite elements, the approach is algebraic and thus applicable to other discretizations in factored form. libCEED's role, as a lightweight portable library that allows a wide variety of applications to share highly optimized discretization kernels, is illustrated in Fig. 1.2, where a non-exhaustive list of specialized implementations (backends) is provided. libCEED provides a low-level Application Programming Interface (API) for user codes so that applications with their own discretization infrastructure (e.g., those in PETSc, MFEM and Nek5000) can evaluate and use the core operations provided by libCEED. GPU implementations are available via pure CUDA and pure HIP as well as the OCCA and MAGMA libraries. CPU implementations are available via pure C and AVX intrinsics as well as the LIBXSMM library. libCEED provides a unified interface, so that users only need to write a single source code and can select the desired specialized implementation at run time. Moreover, each process or thread can instantiate an arbitrary number of backends.

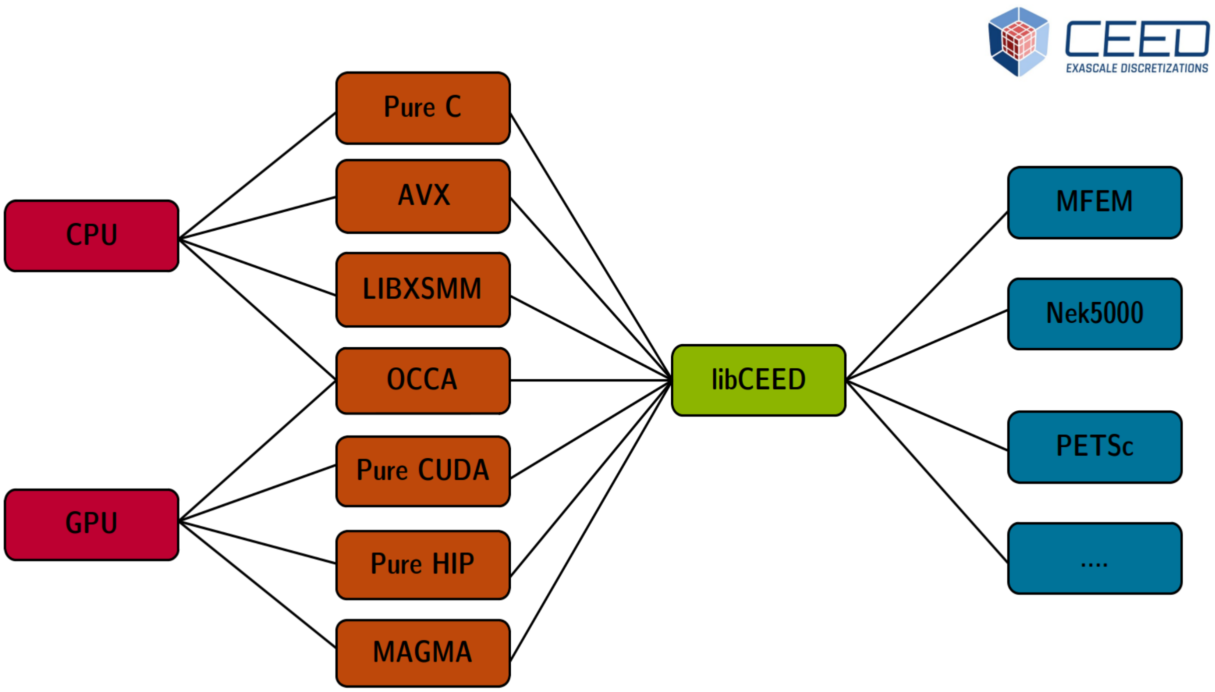


Fig. 1.2: The role of libCEED as a lightweight, portable library which provides a low-level API for efficient, specialized implementations. libCEED allows different applications to share highly optimized discretization kernels.

2 Getting Started

2.1 Building

The CEED library, `libceed`, is a C99 library with no required dependencies, and with Fortran, Python, Julia, and Rust interfaces. It can be built using:

```
$ make
```

or, with optimization flags:

```
$ make OPT='-O3 -march=skylake-avx512 -ffp-contract=fast'
```

These optimization flags are used by all languages (C, C++, Fortran) and this makefile variable can also be set for testing and examples (below).

The library attempts to automatically detect support for the AVX instruction set using gcc-style compiler options for the host. Support may need to be manually specified via:

```
$ make AVX=1
```

or:

```
$ make AVX=0
```

if your compiler does not support gcc-style options, if you are cross compiling, etc.

To enable CUDA support, add `CUDA_DIR=/opt/cuda` or an appropriate directory to your `make` invocation. To enable HIP support, add `ROCM_DIR=/opt/rocm` or an appropriate directory. To enable SYCL support, add `SYCL_DIR=/opt/sycl` or an appropriate directory. Note that SYCL backends require building with oneAPI compilers as well:

```
$ . /opt/intel/oneapi/setvars.sh
$ make SYCL_DIR=/opt/intel/oneapi/compiler/latest/linux SYCLCXX=icpx CC=icx CXX=icpx
```

The library can be configured for host applications which use OpenMP paralellism via:

```
$ make OPENMP=1
```

which will allow operators created and applied from different threads inside an `omp parallel` region.

To store these or other arguments as defaults for future invocations of `make`, use:

```
$ make configure CUDA_DIR=/usr/local/cuda ROCM_DIR=/opt/rocm OPT='-O3 -march=znver2'
```

which stores these variables in `config.mk`.

2.1.1 WebAssembly

libCEED can be built for WASM using Emscripten. For example, one can build the library and run a standalone WASM executable using

```
$ emmake make build/ex2-surface.wasm
$ wasmer build/ex2-surface.wasm -- -s 200000
```

2.2 Additional Language Interfaces

The Fortran interface is built alongside the library automatically.

Python users can install using:

```
$ pip install libceed
```

or in a clone of the repository via `pip install .`.

Julia users can install using:

```
$ julia
julia> ]
pkg> add LibCEED
```

See the LibCEED.jl documentation for more information.

Rust users can include libCEED via `Cargo.toml`:

```
[dependencies]
libceed = "0.12.0"
```

See the Cargo documentation for details.

2.3 Testing

The test suite produces TAP output and is run by:

```
$ make test
```

or, using the `prove` tool distributed with Perl (recommended):

```
$ make prove
```

2.4 Backends

There are multiple supported backends, which can be selected at runtime in the examples:

CEED resource	Backend	Deterministic Capable
CPU Native		
`/cpu/self/ref/serial`	Serial reference implementation	Yes
`/cpu/self/ref/blocked`	Blocked reference implementation	Yes
`/cpu/self/opt/serial`	Serial optimized C implementation	Yes
`/cpu/self/opt/blocked`	Blocked optimized C implementation	Yes
`/cpu/self/avx/serial`	Serial AVX implementation	Yes
`/cpu/self/avx/blocked`	Blocked AVX implementation	Yes
CPU Valgrind		
`/cpu/self/memcheck/*`	Memcheck backends, undefined value checks	Yes
CPU LIBXSMM		
`/cpu/self/xsmm/serial`	Serial LIBXSMM implementation	Yes
`/cpu/self/xsmm/blocked`	Blocked LIBXSMM implementation	Yes
CUDA Native		
`/gpu/cuda/ref`	Reference pure CUDA kernels	Yes
`/gpu/cuda/shared`	Optimized pure CUDA kernels using shared memory	Yes
`/gpu/cuda/gen`	Optimized pure CUDA kernels using code generation	No
HIP Native		
`/gpu/hip/ref`	Reference pure HIP kernels	Yes
`/gpu/hip/shared`	Optimized pure HIP kernels using shared memory	Yes
`/gpu/hip/gen`	Optimized pure HIP kernels using code generation	No
SYCL Native		
`/gpu/sycl/ref`	Reference pure SYCL kernels	Yes
`/gpu/sycl/shared`	Optimized pure SYCL kernels using shared memory	Yes
MAGMA		
`/gpu/cuda/magma`	CUDA MAGMA kernels	No
`/gpu/cuda/magma/det`	CUDA MAGMA kernels	Yes
`/gpu/hip/magma`	HIP MAGMA kernels	No
`/gpu/hip/magma/det`	HIP MAGMA kernels	Yes

continues on next page

Table 2.1 – continued from previous page

CEED resource	Backend	Deterministic Capable
OCCA		
`/*/occa`	Selects backend based on available OCCA modes	Yes
`/cpu/self/occa`	OCCA backend with serial CPU kernels	Yes
`/cpu/openmp/occa`	OCCA backend with OpenMP kernels	Yes
`/cpu/dpcpp/occa`	OCCA backend with DPC++ kernels	Yes
`/gpu/cuda/occa`	OCCA backend with CUDA kernels	Yes
`/gpu/hip/occa`	OCCA backend with HIP kernels	Yes

The `/cpu/self/*/serial` backends process one element at a time and are intended for meshes with a smaller number of high order elements. The `/cpu/self/*/blocked` backends process blocked batches of eight interlaced elements and are intended for meshes with higher numbers of elements.

The `/cpu/self/ref/*` backends are written in pure C and provide basic functionality.

The `/cpu/self/opt/*` backends are written in pure C and use partial e-vectors to improve performance.

The `/cpu/self/avx/*` backends rely upon AVX instructions to provide vectorized CPU performance.

The `/cpu/self/memcheck/*` backends rely upon the Valgrind Memcheck tool to help verify that user QFunctions have no undefined values. To use, run your code with Valgrind and the Memcheck backends, e.g. `valgrind ./build/ex1 -ceed /cpu/self/ref/memcheck`. A 'development' or 'debugging' version of Valgrind with headers is required to use this backend. This backend can be run in serial or blocked mode and defaults to running in the serial mode if `/cpu/self/memcheck` is selected at runtime.

The `/cpu/self/xsmm/*` backends rely upon the LIBXSMM package to provide vectorized CPU performance. If linking MKL and LIBXSMM is desired but the Makefile is not detecting `MKLROOT`, linking libCEED against MKL can be forced by setting the environment variable `MKL=1`.

The `/gpu/cuda/*` backends provide GPU performance strictly using CUDA.

The `/gpu/hip/*` backends provide GPU performance strictly using HIP. They are based on the `/gpu/cuda/*` backends. ROCm version 4.2 or newer is required.

The `/gpu/sycl/*` backends provide GPU performance strictly using SYCL. They are based on the `/gpu/cuda/*` and `/gpu/hip/*` backends.

The `/gpu/*/magma/*` backends rely upon the MAGMA package. To enable the MAGMA backends, the environment variable `MAGMA_DIR` must point to the top-level MAGMA directory, with the MAGMA library located in `$(MAGMA_DIR)/lib/`. By default, `MAGMA_DIR` is set to `../magma`; to build the MAGMA backends with a MAGMA installation located elsewhere, create a link to `magma/` in libCEED's parent directory, or set `MAGMA_DIR` to the proper location. MAGMA version 2.5.0 or newer is required. Currently, each MAGMA library installation is only built for either CUDA or HIP. The corresponding set of libCEED backends (`/gpu/cuda/magma/*` or `/gpu/hip/magma/*`) will automatically be built for the version of the MAGMA library found in `MAGMA_DIR`.

Users can specify a device for all CUDA, HIP, and MAGMA backends through adding `:device_id=#` after the resource name. For example:

- `/gpu/cuda/gen:device_id=1`

The `/*/occa` backends rely upon the OCCA package to provide cross platform performance. To enable the OCCA backend, the environment variable `OCCA_DIR` must point to the top-level OCCA directory, with the OCCA library located in the `${OCCA_DIR}/lib` (By default, `OCCA_DIR` is set to `../occa`). OCCA version 1.4.0 or newer is required.

Users can pass specific OCCA device properties after setting the CEED resource. For example:

- `"/*/occa:mode='CUDA',device_id=0"`

Bit-for-bit reproducibility is important in some applications. However, some libCEED backends use non-deterministic operations, such as `atomicAdd` for increased performance. The backends which are capable of generating reproducible results, with the proper compilation options, are highlighted in the list above.

2.5 Examples

libCEED comes with several examples of its usage, ranging from standalone C codes in the `/examples/ceed` directory to examples based on external packages, such as MFEM, PETSc, and Nek5000. Nek5000 v18.0 or greater is required.

To build the examples, set the `MFEM_DIR`, `PETSC_DIR`, and `NEK5K_DIR` variables and run:

```
$ cd examples/
```

```
# libCEED examples on CPU and GPU
$ cd ceed/
$ make
$ ./ex1-volume -ceed /cpu/self
$ ./ex1-volume -ceed /gpu/cuda
$ ./ex2-surface -ceed /cpu/self
$ ./ex2-surface -ceed /gpu/cuda
$ cd ..

# MFEM+libCEED examples on CPU and GPU
$ cd mfem/
$ make
$ ./bp1 -ceed /cpu/self -no-vis
$ ./bp3 -ceed /gpu/cuda -no-vis
$ cd ..

# Nek5000+libCEED examples on CPU and GPU
$ cd nek/
$ make
$ ./nek-examples.sh -e bp1 -ceed /cpu/self -b 3
$ ./nek-examples.sh -e bp3 -ceed /gpu/cuda -b 3
$ cd ..

# PETSc+libCEED examples on CPU and GPU
$ cd petsc/
$ make
$ ./bps -problem bp1 -ceed /cpu/self
$ ./bps -problem bp2 -ceed /gpu/cuda
$ ./bps -problem bp3 -ceed /cpu/self
$ ./bps -problem bp4 -ceed /gpu/cuda
$ ./bps -problem bp5 -ceed /cpu/self
$ ./bps -problem bp6 -ceed /gpu/cuda
$ cd ..

$ cd petsc/
$ make
$ ./bpsraw -problem bp1 -ceed /cpu/self
$ ./bpsraw -problem bp2 -ceed /gpu/cuda
$ ./bpsraw -problem bp3 -ceed /cpu/self
```

(continues on next page)

(continued from previous page)

```
$ ./bpsraw -problem bp4 -ceed /gpu/cuda
$ ./bpsraw -problem bp5 -ceed /cpu/self
$ ./bpsraw -problem bp6 -ceed /gpu/cuda
$ cd ..

$ cd petsc/
$ make
$ ./bpssphere -problem bp1 -ceed /cpu/self
$ ./bpssphere -problem bp2 -ceed /gpu/cuda
$ ./bpssphere -problem bp3 -ceed /cpu/self
$ ./bpssphere -problem bp4 -ceed /gpu/cuda
$ ./bpssphere -problem bp5 -ceed /cpu/self
$ ./bpssphere -problem bp6 -ceed /gpu/cuda
$ cd ..

$ cd petsc/
$ make
$ ./area -problem cube -ceed /cpu/self -degree 3
$ ./area -problem cube -ceed /gpu/cuda -degree 3
$ ./area -problem sphere -ceed /cpu/self -degree 3 -dm_refine 2
$ ./area -problem sphere -ceed /gpu/cuda -degree 3 -dm_refine 2

$ cd fluids/
$ make
$ ./navierstokes -ceed /cpu/self -degree 1
$ ./navierstokes -ceed /gpu/cuda -degree 1
$ cd ..

$ cd solids/
$ make
$ ./elasticity -ceed /cpu/self -mesh [.exo file] -degree 2 -E 1 -nu 0.3 -problem Linear -forcing mms
$ ./elasticity -ceed /gpu/cuda -mesh [.exo file] -degree 2 -E 1 -nu 0.3 -problem Linear -forcing mms
$ cd ..
```

For the last example shown, sample meshes to be used in place of `[.exo file]` can be found at https://github.com/jeremylt/ceedSampleMeshes

The above code assumes a GPU-capable machine with the CUDA backends enabled. Depending on the available backends, other CEED resource specifiers can be provided with the `-ceed` option. Other command line arguments can be found in examples/petsc.

2.6 Benchmarks

A sequence of benchmarks for all enabled backends can be run using:

```
$ make benchmarks
```

The results from the benchmarks are stored inside the `benchmarks/` directory and they can be viewed using the commands (requires python with matplotlib):

```
$ cd benchmarks
$ python postprocess-plot.py petsc-bps-bp1-*-output.txt
$ python postprocess-plot.py petsc-bps-bp3-*-output.txt
```

Using the `benchmarks` target runs a comprehensive set of benchmarks which may take some time to run. Subsets of the benchmarks can be run using the scripts in the `benchmarks` folder.

For more details about the benchmarks, see the `benchmarks/README.md` file.

2.7 Install

To install libCEED, run:

```
$ make install prefix=/path/to/install/dir
```

or (e.g., if creating packages):

```
$ make install prefix=/usr DESTDIR=/packaging/path
```

To build and install in separate steps, run:

```
$ make for_install=1 prefix=/path/to/install/dir
$ make install prefix=/path/to/install/dir
```

The usual variables like `CC` and `CFLAGS` are used, and optimization flags for all languages can be set using the likes of `OPT='-O3 -march=native'`. Use `STATIC=1` to build static libraries (`libceed.a`).

To install libCEED for Python, run:

```
$ pip install libceed
```

with the desired setuptools options, such as `--user`.

2.7.1 pkg-config

In addition to library and header, libCEED provides a pkg-config file that can be used to easily compile and link. For example, if `$prefix` is a standard location or you set the environment variable `PKG_CONFIG_PATH`:

```
$ cc `pkg-config --cflags --libs ceed` -o myapp myapp.c
```

will build `myapp` with libCEED. This can be used with the source or installed directories. Most build systems have support for pkg-config.

2.8 Contact

You can reach the libCEED team by emailing ceed-users@llnl.gov or by leaving a comment in the issue tracker.

2.9 How to Cite

If you utilize libCEED please cite:

```
@article{libceed-joss-paper,
  author       = {Jed Brown and Ahmad Abdelfattah and Valeria Barra and Natalie Beams␣
↪and Jean Sylvain Camier and Veselin Dobrev and Yohann Dudouit and Leila Ghaffari␣
↪and Tzanio Kolev and David Medina and Will Pazner and Thilina Ratnayaka and Jeremy␣
↪Thompson and Stan Tomov},
  title        = {{libCEED}: Fast algebra for high-order element-based␣
↪discretizations},
  journal      = {Journal of Open Source Software},
  year         = {2021},
  publisher    = {The Open Journal},
  volume       = {6},
  number       = {63},
  pages        = {2945},
  doi          = {10.21105/joss.02945}
}
```

The archival copy of the libCEED user manual is maintained on Zenodo. To cite the user manual:

```
@misc{libceed-user-manual,
  author       = {Abdelfattah, Ahmad and
                  Barra, Valeria and
                  Beams, Natalie and
                  Brown, Jed and
                  Camier, Jean-Sylvain and
                  Dobrev, Veselin and
                  Dudouit, Yohann and
                  Ghaffari, Leila and
                  Kolev, Tzanio and
                  Medina, David and
                  Pazner, Will and
                  Ratnayaka, Thilina and
                  Shakeri, Rezgar and
                  Thompson, Jeremy L and
                  Tomov, Stanimire and
                  Wright III, James},
  title        = {{libCEED} User Manual},
  month        = dec,
  year         = 2022,
  publisher    = {Zenodo},
  version      = {0.11.0},
  doi          = {10.5281/zenodo.7480454}
}
```

For libCEED's Python interface please cite:

```
@InProceedings{libceed-paper-proc-scipy-2020,
  author    = {{V}aleria {B}arra and {J}ed {B}rown and {J}eremy {T}hompson and {Y}
↪ohann {D}udouit},
  title     = {{H}igh-performance operator evaluations with ease of use: lib{C}{E}{E}
↪{D}'s {P}ython interface},
  booktitle = {{P}roceedings of the 19th {P}ython in {S}cience {C}onference},
  pages     = {85 - 90},
  year      = {2020},
  editor    = {{M}eghann {A}garwal and {C}hris {C}alloway and {D}illon {N}iederhut␣
```

(continues on next page)

(continued from previous page)

```
 ↪and {D}avid {S}hupe},
  doi       = {10.25080/Majora-342d178e-00c}
}
```

The BibTeX entries for these references can be found in the `doc/bib/references.bib` file.

2.10 Copyright

The following copyright applies to each file in the CEED software suite, unless otherwise stated in the file:

> Copyright (c) 2017-2023, Lawrence Livermore National Security, LLC and other CEED contributors. All rights reserved.

See files LICENSE and NOTICE for details.

3 Interface Concepts

This page provides a brief description of the theoretical foundations and the practical implementation of the libCEED library.

3.1 Theoretical Framework

In finite element formulations, the weak form of a Partial Differential Equation (PDE) is evaluated on a subdomain Ω_e (element) and the local results are composed into a larger system of equations that models the entire problem on the global domain Ω. In particular, when high-order finite elements or spectral elements are used, the resulting sparse matrix representation of the global operator is computationally expensive, with respect to both the memory transfer and floating point operations needed for its evaluation. libCEED provides an interface for matrix-free operator description that enables efficient evaluation on a variety of computational device types (selectable at run time). We present here the notation and the mathematical formulation adopted in libCEED.

We start by considering the discrete residual $F(u) = 0$ formulation in weak form. We first define the L^2 inner product between real-valued functions

$$\langle v, u \rangle = \int_\Omega v u d\boldsymbol{x},$$

where $\boldsymbol{x} \in \mathbb{R}^d \supset \Omega$.

We want to find u in a suitable space V_D, such that

$$\langle \boldsymbol{v}, \boldsymbol{f}(u) \rangle = \int_\Omega \boldsymbol{v} \cdot \boldsymbol{f}_0(u, \nabla u) + \nabla \boldsymbol{v} : \boldsymbol{f}_1(u, \nabla u) = 0 \tag{3.1}$$

for all $\boldsymbol{v}$ in the corresponding homogeneous space V_0, where $\boldsymbol{f}_0$ and $\boldsymbol{f}_1$ contain all possible sources in the problem. We notice here that $\boldsymbol{f}_0$ represents all terms in (3.1) which multiply the (possibly vector-valued) test function $\boldsymbol{v}$ and $\boldsymbol{f}_1$ all terms which multiply its gradient $\nabla \boldsymbol{v}$. For an n-component problems in d dimensions, $\boldsymbol{f}_0 \in \mathbb{R}^n$ and $\boldsymbol{f}_1 \in \mathbb{R}^{nd}$.

Note: The notation $\nabla \boldsymbol{v} \colon \boldsymbol{f}_1$ represents contraction over both fields and spatial dimensions while a single dot represents contraction in just one, which should be clear from context, e.g., $\boldsymbol{v} \cdot \boldsymbol{f}_0$ contracts only over fields.

Note: In the code, the function that represents the weak form at quadrature points is called the *CeedQFunction*. In the *Examples* provided with the library (in the `examples/` directory), we store the term f_0 directly into `v`, and the term f_1 directly into `dv` (which stands for ∇v). If equation (3.1) only presents a term of the type f_0, the *CeedQFunction* will only have one output argument, namely `v`. If equation (3.1) also presents a term of the type f_1, then the *CeedQFunction* will have two output arguments, namely, `v` and `dv`.

3.2 Finite Element Operator Decomposition

Finite element operators are typically defined through weak formulations of partial differential equations that involve integration over a computational mesh. The required integrals are computed by splitting them as a sum over the mesh elements, mapping each element to a simple *reference* element (e.g. the unit square) and applying a quadrature rule in reference space.

This sequence of operations highlights an inherent hierarchical structure present in all finite element operators where the evaluation starts on *global (trial) degrees of freedom (DoFs) or nodes on the whole mesh*, restricts to *DoFs on subdomains* (groups of elements), then moves to independent *DoFs on each element*, transitions to independent *quadrature points* in reference space, performs the integration, and then goes back in reverse order to global (test) degrees of freedom on the whole mesh.

This is illustrated below for the simple case of symmetric linear operator on third order (Q_3) scalar continuous (H^1) elements, where we use the notions **T-vector**, **L-vector**, **E-vector** and **Q-vector** to represent the sets corresponding to the (true) degrees of freedom on the global mesh, the split local degrees of freedom on the subdomains, the split degrees of freedom on the mesh elements, and the values at quadrature points, respectively.

We refer to the operators that connect the different types of vectors as:

- Subdomain restriction $\boldsymbol{P}$
- Element restriction $\mathcal{E}$
- Basis (Dofs-to-Qpts) evaluator $\boldsymbol{B}$
- Operator at quadrature points $\boldsymbol{D}$

More generally, when the test and trial space differ, they get their own versions of $\boldsymbol{P}$, $\mathcal{E}$ and $\boldsymbol{B}$.

Note that in the case of adaptive mesh refinement (AMR), the restrictions $\boldsymbol{P}$ and $\mathcal{E}$ will involve not just extracting sub-vectors, but evaluating values at constrained degrees of freedom through the AMR interpolation. There can also be several levels of subdomains ($\boldsymbol{P}_1$, $\boldsymbol{P}_2$, etc.), and it may be convenient to split $\boldsymbol{D}$ as the product of several operators ($\boldsymbol{D}_1$, $\boldsymbol{D}_2$, etc.).

3.2.1 Terminology and Notation

Vector representation/storage categories:

- True degrees of freedom/unknowns, **T-vector**:
  - each unknown i has exactly one copy, on exactly one processor, $rank(i)$
  - this is a non-overlapping vector decomposition
  - usually includes any essential (fixed) DoFs.

$$A = \mathcal{P}^T \mathcal{E}^T B^T D B \mathcal{E} \mathcal{P}$$

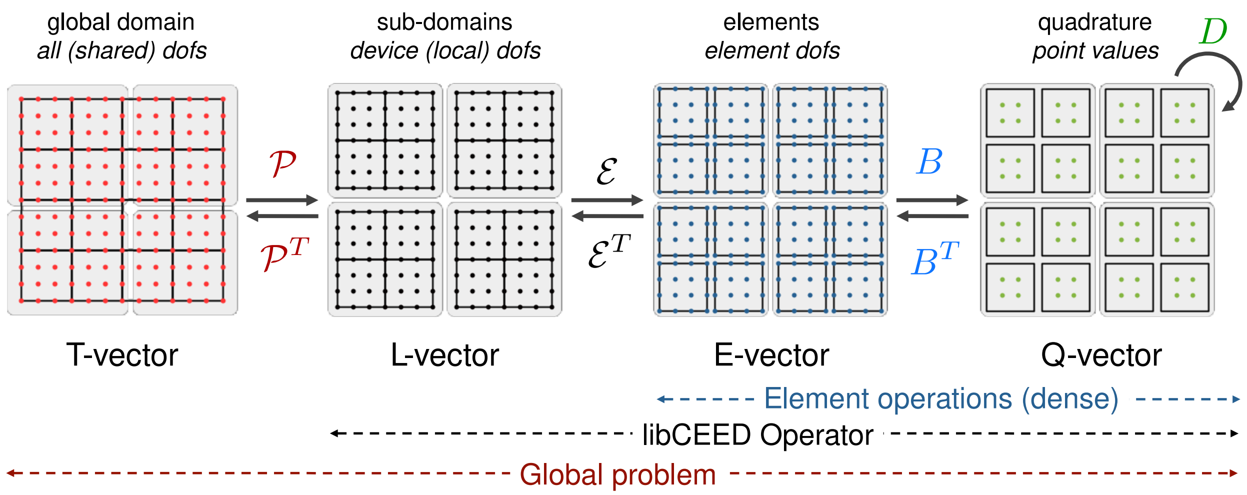
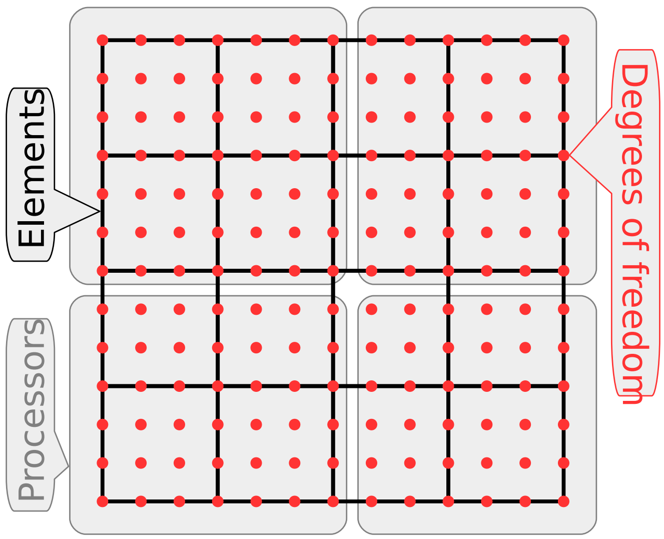
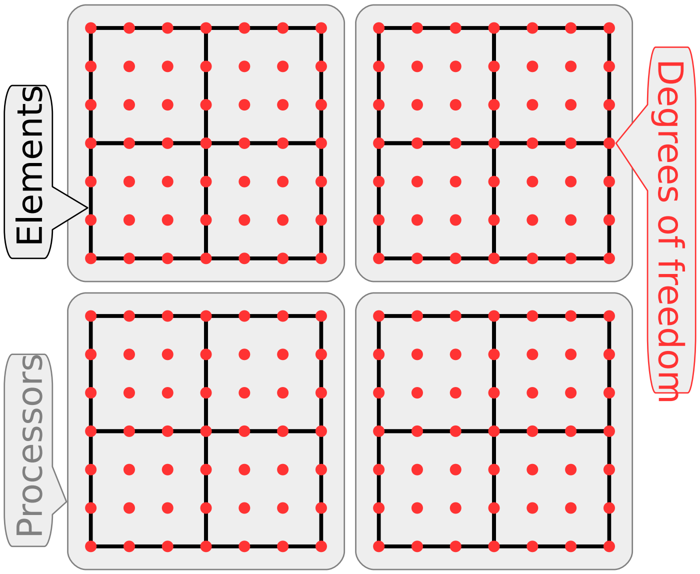


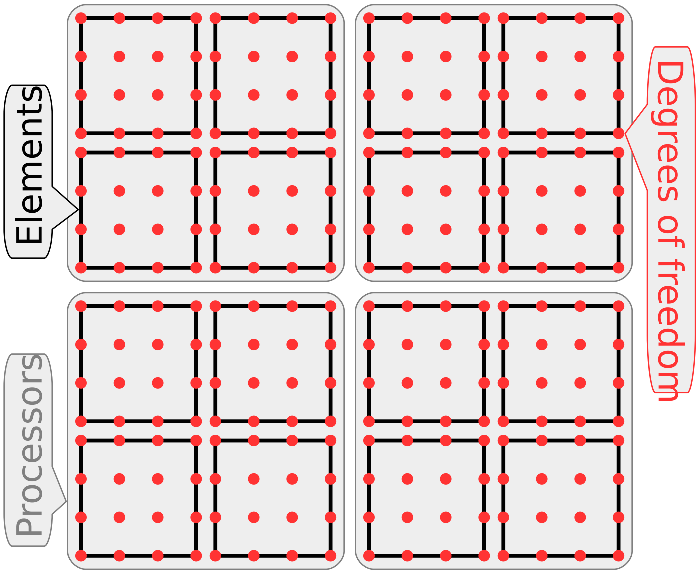
Fig. 3.1: Operator Decomposition



- Local (w.r.t. processors) degrees of freedom/unknowns, **L-vector**:
  - each unknown *i* has exactly one copy on each processor that owns an element containing *i*
  - this is an overlapping vector decomposition with overlaps only across different processors—there is no duplication of unknowns on a single processor
  - the shared DoFs/unknowns are the overlapping DoFs, i.e. the ones that have more than one copy, on different processors.

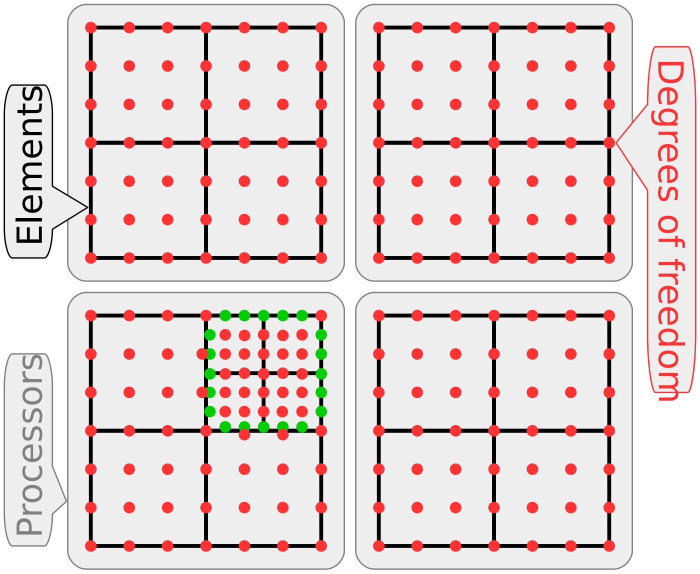


- Per element decomposition, **E-vector**:
    - each unknown *i* has as many copies as the number of elements that contain *i*
    - usually, the copies of the unknowns are grouped by the element they belong to.



- In the case of AMR with hanging nodes (giving rise to hanging DoFs):
    - the **L-vector** is enhanced with the hanging/dependent DoFs
    - the additional hanging/dependent DoFs are duplicated when they are shared by multiple processors
    - this way, an **E-vector** can be derived from an **L-vector** without any communications and without additional computations to derive the dependent DoFs
    - in other words, an entry in an **E-vector** is obtained by copying an entry from the

corresponding **L-vector**, optionally switching the sign of the entry (for $H(\text{div})$—and $H(\text{curl})$-conforming spaces).



- In the case of variable order spaces:
  - the dependent DoFs (usually on the higher-order side of a face/edge) can be treated just like the hanging/dependent DoFs case.
- Quadrature point vector, **Q-vector**:
  - this is similar to **E-vector** where instead of DoFs, the vector represents values at quadrature points, grouped by element.
- In many cases it is useful to distinguish two types of vectors:
  - **X-vector**, or **primal X-vector**, and **X'-vector**, or **dual X-vector**
  - here X can be any of the T, L, E, or Q categories
  - for example, the mass matrix operator maps a **T-vector** to a **T'-vector**
  - the solutions vector is a **T-vector**, and the RHS vector is a **T'-vector**
  - using the parallel prolongation operator, one can map the solution **T-vector** to a solution **L-vector**, etc.

Operator representation/storage/action categories:

- Full true-DoF parallel assembly, **TA**, or **A**:
  - ParCSR or similar format
  - the T in TA indicates that the data format represents an operator from a **T-vector** to a **T'-vector**.
- Full local assembly, **LA**:
  - CSR matrix on each rank
  - the parallel prolongation operator, ***P***, (and its transpose) should use optimized matrix-free action
  - note that ***P*** is the operator mapping T-vectors to L-vectors.
- Element matrix assembly, **EA**:

  - each element matrix is stored as a dense matrix
  - optimized element and parallel prolongation operators
  - note that the element prolongation operator is the mapping from an **L-vector** to an **E-vector**.
- Quadrature-point/partial assembly, **QA** or **PA**:
  - precompute and store $w \det(J)$ at all quadrature points in all mesh elements
  - the stored data can be viewed as a **Q-vector**.
- Unassembled option, **UA** or **U**:
  - no assembly step
  - the action uses directly the mesh node coordinates, and assumes specific form of the coefficient, e.g. constant, piecewise-constant, or given as a **Q-vector** (Q-coefficient).

3.2.2 Partial Assembly

Since the global operator $\boldsymbol{A}$ is just a series of variational restrictions with $\boldsymbol{B}$, $\mathcal{E}$ and $\boldsymbol{P}$, starting from its point-wise kernel $\boldsymbol{D}$, a "matvec" with $\boldsymbol{A}$ can be performed by evaluating and storing some of the innermost variational restriction matrices, and applying the rest of the operators "on-the-fly". For example, one can compute and store a global matrix on **T-vector** level. Alternatively, one can compute and store only the subdomain (**L-vector**) or element (**E-vector**) matrices and perform the action of $\boldsymbol{A}$ using matvecs with $\boldsymbol{P}$ or $\boldsymbol{P}$ and $\mathcal{E}$. While these options are natural for low-order discretizations, they are not a good fit for high-order methods due to the amount of FLOPs needed for their evaluation, as well as the memory transfer needed for a matvec.

Our focus in libCEED, instead, is on **partial assembly**, where we compute and store only $\boldsymbol{D}$ (or portions of it) and evaluate the actions of $\boldsymbol{P}$, $\mathcal{E}$ and $\boldsymbol{B}$ on-the-fly. Critically for performance, we take advantage of the tensor-product structure of the degrees of freedom and quadrature points on *quad* and *hex* elements to perform the action of $\boldsymbol{B}$ without storing it as a matrix.

Implemented properly, the partial assembly algorithm requires optimal amount of memory transfers (with respect to the polynomial order) and near-optimal FLOPs for operator evaluation. It consists of an operator *setup* phase, that evaluates and stores $\boldsymbol{D}$ and an operator *apply* (evaluation) phase that computes the action of $\boldsymbol{A}$ on an input vector. When desired, the setup phase may be done as a side-effect of evaluating a different operator, such as a nonlinear residual. The relative costs of the setup and apply phases are different depending on the physics being expressed and the representation of $\boldsymbol{D}$.

3.2.3 Parallel Decomposition

After the application of each of the first three transition operators, $\boldsymbol{P}$, $\mathcal{E}$ and $\boldsymbol{B}$, the operator evaluation is decoupled on their ranges, so $\boldsymbol{P}$, $\mathcal{E}$ and $\boldsymbol{B}$ allow us to "zoom-in" to subdomain, element and quadrature point level, ignoring the coupling at higher levels.

Thus, a natural mapping of $\boldsymbol{A}$ on a parallel computer is to split the **T-vector** over MPI ranks (a non-overlapping decomposition, as is typically used for sparse matrices), and then split the rest of the vector types over computational devices (CPUs, GPUs, etc.) as indicated by the shaded regions in the diagram above.

One of the advantages of the decomposition perspective in these settings is that the operators $\boldsymbol{P}$, $\mathcal{E}$, $\boldsymbol{B}$ and $\boldsymbol{D}$ clearly separate the MPI parallelism in the operator ($\boldsymbol{P}$) from the unstructured mesh topology ($\mathcal{E}$), the choice of the finite element space/basis ($\boldsymbol{B}$) and the geometry and point-wise physics $\boldsymbol{D}$. These components also naturally fall in different classes of numerical algorithms – parallel (multi-device) linear algebra for

P, sparse (on-device) linear algebra for $\mathcal{E}$, dense/structured linear algebra (tensor contractions) for ***B*** and parallel point-wise evaluations for ***D***.

Currently in libCEED, it is assumed that the host application manages the global **T-vectors** and the required communications among devices (which are generally on different compute nodes) with **P**. Our API is thus focused on the **L-vector** level, where the logical devices, which in the library are represented by the *Ceed* object, are independent. Each MPI rank can use one or more *Ceed*s, and each *Ceed*, in turn, can represent one or more physical devices, as long as libCEED backends support such configurations. The idea is that every MPI rank can use any logical device it is assigned at runtime. For example, on a node with 2 CPU sockets and 4 GPUs, one may decide to use 6 MPI ranks (each using a single *Ceed* object): 2 ranks using 1 CPU socket each, and 4 using 1 GPU each. Another choice could be to run 1 MPI rank on the whole node and use 5 *Ceed* objects: 1 managing all CPU cores on the 2 sockets and 4 managing 1 GPU each. The communications among the devices, e.g. required for applying the action of ***P***, are currently out of scope of libCEED. The interface is non-blocking for all operations involving more than O(1) data, allowing operations performed on a coprocessor or worker threads to overlap with operations on the host.

3.3 API Description

The libCEED API takes an algebraic approach, where the user essentially describes in the *frontend* the operators $\mathcal{E}$, ***B***, and ***D*** and the library provides *backend* implementations and coordinates their action to the original operator on **L-vector** level (i.e. independently on each device / MPI task). This is visualized in the schematic below; "active" and "passive" inputs/outputs will be discussed in more detail later.

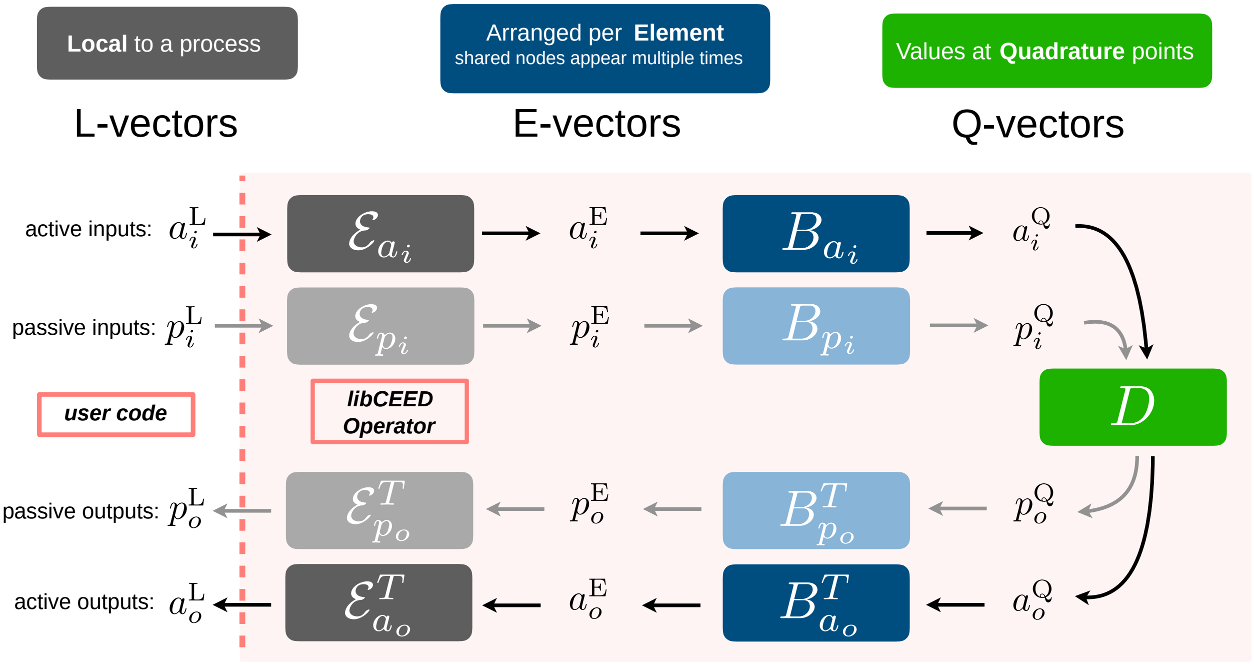


Fig. 3.2: Flow of data through vector types inside libCEED Operators, through backend implementations of $\mathcal{E}$, ***B***, and ***D***

One of the advantages of this purely algebraic description is that it already includes all the finite element information, so the backends can operate on linear algebra level without explicit finite element code. The frontend description is general enough to support a wide variety of finite element algorithms, as well as some other types algorithms such as spectral finite differences. The separation of the front- and backends enables applications to easily switch/try different backends. It also enables backend developers to impact many applications from a single implementation.

Our long-term vision is to include a variety of backend implementations in libCEED, ranging from reference kernels to highly optimized kernels targeting specific devices (e.g. GPUs) or specific polynomial orders. A simple reference backend implementation is provided in the file ceed-ref.c.

On the frontend, the mapping between the decomposition concepts and the code implementation is as follows:

- **L-**, **E-** and **Q-vector** are represented as variables of type *CeedVector*. (A backend may choose to operate incrementally without forming explicit **E-** or **Q-vectors**.)
- $\mathcal{E}$ is represented as variable of type *CeedElemRestriction*.
- $\boldsymbol{B}$ is represented as variable of type *CeedBasis*.
- the action of $\boldsymbol{D}$ is represented as variable of type *CeedQFunction*.
- the overall operator $\mathcal{E}^T \boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B} \mathcal{E}$ is represented as variable of type *CeedOperator* and its action is accessible through `CeedOperatorApply()`.

To clarify these concepts and illustrate how they are combined in the API, consider the implementation of the action of a simple 1D mass matrix (cf. tests/t500-operator.c).

```c
/// @file
/// Test creation, action, and destruction for mass matrix operator
/// \test Test creation, action, and destruction for mass matrix operator
#include "t500-operator.h"

#include <ceed.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  Ceed                ceed;
  CeedElemRestriction elem_restriction_x, elem_restriction_u, elem_restriction_q_data;
  CeedBasis           basis_x, basis_u;
  CeedQFunction       qf_setup, qf_mass;
  CeedOperator        op_setup, op_mass;
  CeedVector          q_data, x, u, v;
  CeedInt             num_elem = 15, p = 5, q = 8;
  CeedInt             num_nodes_x = num_elem + 1, num_nodes_u = num_elem * (p - 1) + 1;
  CeedInt             ind_x[num_elem * 2], ind_u[num_elem * p];
  CeedScalar          x_array[num_nodes_x];

  //! [Ceed Init]
  CeedInit(argv[1], &ceed);
  //! [Ceed Init]
  for (CeedInt i = 0; i < num_nodes_x; i++) x_array[i] = (CeedScalar)i / (num_nodes_x - 1);
  for (CeedInt i = 0; i < num_elem; i++) {
    ind_x[2 * i + 0] = i;
    ind_x[2 * i + 1] = i + 1;
  }
  //! [ElemRestr Create]
  CeedElemRestrictionCreate(ceed, num_elem, 2, 1, 1, num_nodes_x, CEED_MEM_HOST, CEED_USE_POINTER, ind_x, &elem_restriction_x);
  //! [ElemRestr Create]

  for (CeedInt i = 0; i < num_elem; i++) {
```

(continues on next page)

(continued from previous page)

```
    for (CeedInt j = 0; j < p; j++) {
      ind_u[p * i + j] = i * (p - 1) + j;
    }
  }
  //! [ElemRestrU Create]
  CeedElemRestrictionCreate(ceed, num_elem, p, 1, 1, num_nodes_u, CEED_MEM_HOST, CEED_USE_POINTER, ind_u, &elem_restriction_u);
  CeedInt strides_q_data[3] = {1, q, q};
  CeedElemRestrictionCreateStrided(ceed, num_elem, q, 1, q * num_elem, strides_q_data, &elem_restriction_q_data);
  //! [ElemRestrU Create]

  //! [Basis Create]
  CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, 2, q, CEED_GAUSS, &basis_x);
  CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, p, q, CEED_GAUSS, &basis_u);
  //! [Basis Create]

  //! [QFunction Create]
  CeedQFunctionCreateInterior(ceed, 1, setup, setup_loc, &qf_setup);
  CeedQFunctionAddInput(qf_setup, "weight", 1, CEED_EVAL_WEIGHT);
  CeedQFunctionAddInput(qf_setup, "dx", 1, CEED_EVAL_GRAD);
  CeedQFunctionAddOutput(qf_setup, "rho", 1, CEED_EVAL_NONE);

  CeedQFunctionCreateInterior(ceed, 1, mass, mass_loc, &qf_mass);
  CeedQFunctionAddInput(qf_mass, "rho", 1, CEED_EVAL_NONE);
  CeedQFunctionAddInput(qf_mass, "u", 1, CEED_EVAL_INTERP);
  CeedQFunctionAddOutput(qf_mass, "v", 1, CEED_EVAL_INTERP);
  //! [QFunction Create]

  //! [Setup Create]
  CeedOperatorCreate(ceed, qf_setup, CEED_QFUNCTION_NONE, CEED_QFUNCTION_NONE, &op_setup);
  //! [Setup Create]

  //! [Operator Create]
  CeedOperatorCreate(ceed, qf_mass, CEED_QFUNCTION_NONE, CEED_QFUNCTION_NONE, &op_mass);
  //! [Operator Create]

  CeedVectorCreate(ceed, num_nodes_x, &x);
  CeedVectorSetArray(x, CEED_MEM_HOST, CEED_USE_POINTER, x_array);
  CeedVectorCreate(ceed, num_elem * q, &q_data);

  //! [Setup Set]
  CeedOperatorSetField(op_setup, "weight", CEED_ELEMRESTRICTION_NONE, basis_x, CEED_VECTOR_NONE);
  CeedOperatorSetField(op_setup, "dx", elem_restriction_x, basis_x, CEED_VECTOR_ACTIVE);
  CeedOperatorSetField(op_setup, "rho", elem_restriction_q_data, CEED_BASIS_NONE, CEED_VECTOR_ACTIVE);
  //! [Setup Set]

  //! [Operator Set]
  CeedOperatorSetField(op_mass, "rho", elem_restriction_q_data, CEED_BASIS_NONE, q_data);
  CeedOperatorSetField(op_mass, "u", elem_restriction_u, basis_u, CEED_VECTOR_ACTIVE);
```

(continues on next page)

(continued from previous page)

```
  CeedOperatorSetField(op_mass, "v", elem_restriction_u, basis_u, CEED_VECTOR_ACTIVE);
  //! [Operator Set]

  //! [Setup Apply]
  CeedOperatorApply(op_setup, x, q_data, CEED_REQUEST_IMMEDIATE);
  //! [Setup Apply]

  CeedVectorCreate(ceed, num_nodes_u, &u);
  CeedVectorSetValue(u, 0.0);
  CeedVectorCreate(ceed, num_nodes_u, &v);
  //! [Operator Apply]
  CeedOperatorApply(op_mass, u, v, CEED_REQUEST_IMMEDIATE);
  //! [Operator Apply]

  {
    const CeedScalar *v_array;

    CeedVectorGetArrayRead(v, CEED_MEM_HOST, &v_array);
    for (CeedInt i = 0; i < num_nodes_u; i++) {
      if (fabs(v_array[i]) > 1e-14) printf("[%" CeedInt_FMT "] v %g != 0.0\n", i, v_array[i]);
    }
    CeedVectorRestoreArrayRead(v, &v_array);
  }

  CeedVectorDestroy(&x);
  CeedVectorDestroy(&u);
  CeedVectorDestroy(&v);
  CeedVectorDestroy(&q_data);
  CeedElemRestrictionDestroy(&elem_restriction_x);
  CeedElemRestrictionDestroy(&elem_restriction_u);
  CeedElemRestrictionDestroy(&elem_restriction_q_data);
  CeedBasisDestroy(&basis_x);
  CeedBasisDestroy(&basis_u);
  CeedQFunctionDestroy(&qf_setup);
  CeedQFunctionDestroy(&qf_mass);
  CeedOperatorDestroy(&op_setup);
  CeedOperatorDestroy(&op_mass);
  CeedDestroy(&ceed);
  return 0;
}
```

In the following figure, we specialize the schematic used above for general operators so that it corresponds to the specific setup and mass operators as implemented in the sample code. We show that the active output of the setup operator, combining the quadrature weights with the Jacobian information for the mesh transformation, becomes a passive input to the mass operator. Notations denote the libCEED function used to set the properties of the input and output fields.

The constructor

```
  CeedInit(argv[1], &ceed);
```

creates a logical device `ceed` on the specified *resource*, which could also be a coprocessor such as `"/nvidia/0"`. There can be any number of such devices, including multiple logical devices driving the same resource (though performance may suffer in case of oversubscription). The resource is used to locate a suitable backend which will have discretion over the implementations of all objects created with this logical device.

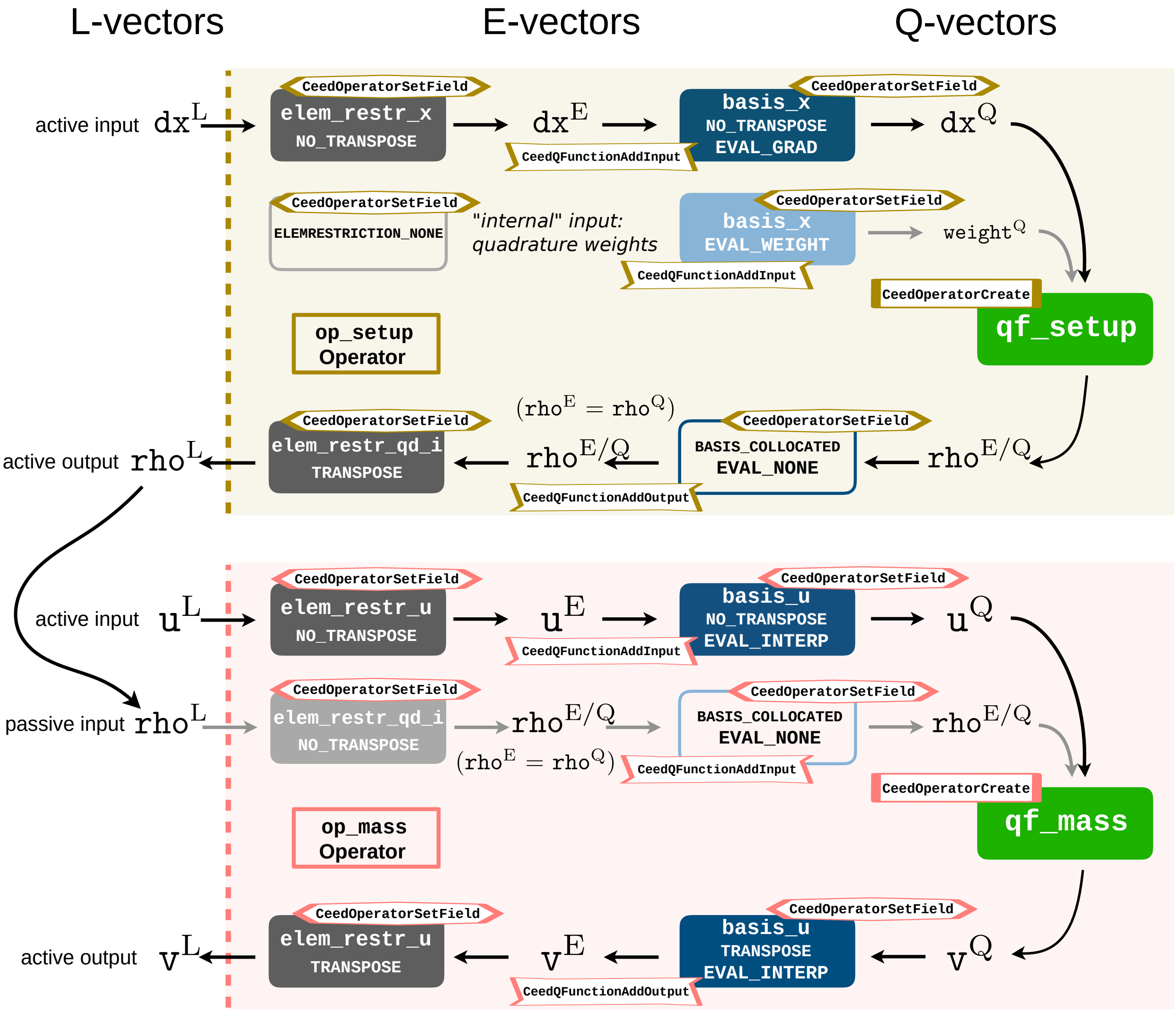


Fig. 3.3: Specific combination of $\mathcal{E}$, $\boldsymbol{B}$, $\boldsymbol{D}$, and input/output vectors corresponding to the libCEED operators in the t500-operator test

The setup routine above computes and stores ***D***, in this case a scalar value in each quadrature point, while mass uses these saved values to perform the action of ***D***. These functions are turned into the *CeedQFunction* variables qf_setup and qf_mass in the *CeedQFunctionCreateInterior()* calls:

```
CeedQFunctionCreateInterior(ceed, 1, setup, setup_loc, &qf_setup);
CeedQFunctionAddInput(qf_setup, "weight", 1, CEED_EVAL_WEIGHT);
CeedQFunctionAddInput(qf_setup, "dx", 1, CEED_EVAL_GRAD);
CeedQFunctionAddOutput(qf_setup, "rho", 1, CEED_EVAL_NONE);

CeedQFunctionCreateInterior(ceed, 1, mass, mass_loc, &qf_mass);
CeedQFunctionAddInput(qf_mass, "rho", 1, CEED_EVAL_NONE);
CeedQFunctionAddInput(qf_mass, "u", 1, CEED_EVAL_INTERP);
CeedQFunctionAddOutput(qf_mass, "v", 1, CEED_EVAL_INTERP);
```

A *CeedQFunction* performs independent operations at each quadrature point and the interface is intended to facilitate vectorization. The second argument is an expected vector length. If greater than 1, the caller must ensure that the number of quadrature points Q is divisible by the vector length. This is often satisfied automatically due to the element size or by batching elements together to facilitate vectorization in other stages, and can always be ensured by padding.

In addition to the function pointers (setup and mass), *CeedQFunction* constructors take a string representation specifying where the source for the implementation is found. This is used by backends that support Just-In-Time (JIT) compilation (i.e., CUDA and OCCA) to compile for coprocessors. For full support across all backends, these *CeedQFunction* source files must only contain constructs mutually supported by C99, C++11, and CUDA. For example, explicit type casting of void pointers and explicit use of compatible arguments for math library functions is required, and variable-length array (VLA) syntax for array reshaping is only available via libCEED's CEED_Q_VLA macro.

Different input and output fields are added individually, specifying the field name, size of the field, and evaluation mode.

The size of the field is provided by a combination of the number of components the effect of any basis evaluations.

The evaluation mode (see *Typedefs and Enumerations*) CEED_EVAL_INTERP for both input and output fields indicates that the mass operator only contains terms of the form

$$\int_\Omega v \cdot f_0(u, \nabla u)$$

where v are test functions (see the *Theoretical Framework*). More general operators, such as those of the form

$$\int_\Omega v \cdot f_0(u, \nabla u) + \nabla v : f_1(u, \nabla u)$$

can be expressed.

For fields with derivatives, such as with the basis evaluation mode (see *Typedefs and Enumerations*) CEED_EVAL_GRAD, the size of the field needs to reflect both the number of components and the geometric dimension. A 3-dimensional gradient on four components would therefore mean the field has a size of 12.

The ***B*** operators for the mesh nodes, basis_x, and the unknown field, basis_u, are defined in the calls to the function *CeedBasisCreateTensorH1Lagrange()*. In this example, both the mesh and the unknown field use H^1 Lagrange finite elements of order 1 and 4 respectively (the P argument represents the number of 1D degrees of freedom on each element). Both basis operators use the same integration rule, which is Gauss-Legendre with 8 points (the Q argument).

```
CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, 2, q, CEED_GAUSS, &basis_x);
CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, p, q, CEED_GAUSS, &basis_u);
```

Other elements with this structure can be specified in terms of the Q×P matrices that evaluate values and gradients at quadrature points in one dimension using *CeedBasisCreateTensorH1()*. Elements that do not have tensor product structure, such as symmetric elements on simplices, will be created using different constructors.

The $\mathcal{E}$ operators for the mesh nodes, `elem_restr_x`, and the unknown field, `elem_restr_u`, are specified in the *CeedElemRestrictionCreate()*. Both of these specify directly the DoF indices for each element in the `ind_x` and `ind_u` arrays:

```
CeedElemRestrictionCreate(ceed, num_elem, 2, 1, 1, num_nodes_x, CEED_MEM_HOST, CEED_
↪USE_POINTER, ind_x, &elem_restriction_x);
```

```
CeedElemRestrictionCreate(ceed, num_elem, p, 1, 1, num_nodes_u, CEED_MEM_HOST, CEED_
↪USE_POINTER, ind_u, &elem_restriction_u);
CeedInt strides_q_data[3] = {1, q, q};
CeedElemRestrictionCreateStrided(ceed, num_elem, q, 1, q * num_elem, strides_q_data,
↪ &elem_restriction_q_data);
```

If the user has arrays available on a device, they can be provided using `CEED_MEM_DEVICE`. This technique is used to provide no-copy interfaces in all contexts that involve problem-sized data.

For discontinuous Galerkin and for applications such as Nek5000 that only explicitly store **E-vectors** (inter-element continuity has been subsumed by the parallel restriction $\boldsymbol{P}$), the element restriction $\mathcal{E}$ is the identity and *CeedElemRestrictionCreateStrided()* is used instead. We plan to support other structured representations of $\mathcal{E}$ which will be added according to demand. There are two common approaches for supporting non-conforming elements: applying the node constraints via $\boldsymbol{P}$ so that the **L-vector** can be processed uniformly and applying the constraints via $\mathcal{E}$ so that the **E-vector** is uniform. The former can be done with the existing interface while the latter will require a generalization to element restriction that would define field values at constrained nodes as linear combinations of the values at primary nodes.

These operations, $\mathcal{E}$, $\boldsymbol{B}$, and $\boldsymbol{D}$, are combined with a *CeedOperator*. As with *CeedQFunction*s, operator fields are added separately with a matching field name, basis ($\boldsymbol{B}$), element restriction ($\mathcal{E}$), and **L-vector**. The flag `CEED_VECTOR_ACTIVE` indicates that the vector corresponding to that field will be provided to the operator when *CeedOperatorApply()* is called. Otherwise the input/output will be read from/written to the specified **L-vector**.

With partial assembly, we first perform a setup stage where $\boldsymbol{D}$ is evaluated and stored. This is accomplished by the operator `op_setup` and its application to `X`, the nodes of the mesh (these are needed to compute Jacobians at quadrature points). Note that the corresponding *CeedOperatorApply()* has no basis evaluation on the output, as the quadrature data is not needed at the DoFs:

```
CeedOperatorCreate(ceed, qf_setup, CEED_QFUNCTION_NONE, CEED_QFUNCTION_NONE, &op_
↪setup);
```

```
CeedOperatorSetField(op_setup, "weight", CEED_ELEMRESTRICTION_NONE, basis_x, CEED_
↪VECTOR_NONE);
CeedOperatorSetField(op_setup, "dx", elem_restriction_x, basis_x, CEED_VECTOR_
↪ACTIVE);
CeedOperatorSetField(op_setup, "rho", elem_restriction_q_data, CEED_BASIS_NONE,␣
↪CEED_VECTOR_ACTIVE);
```

```
CeedOperatorApply(op_setup, x, q_data, CEED_REQUEST_IMMEDIATE);
```

The action of the operator is then represented by operator `op_mass` and its *CeedOperatorApply()* to the input **L-vector** `U` with output in `V`:

```
  CeedOperatorCreate(ceed, qf_mass, CEED_QFUNCTION_NONE, CEED_QFUNCTION_NONE, &op_mass);
```

```
  CeedOperatorSetField(op_mass, "rho", elem_restriction_q_data, CEED_BASIS_NONE, q_data);
  CeedOperatorSetField(op_mass, "u", elem_restriction_u, basis_u, CEED_VECTOR_ACTIVE);
  CeedOperatorSetField(op_mass, "v", elem_restriction_u, basis_u, CEED_VECTOR_ACTIVE);
```

```
  CeedOperatorApply(op_mass, u, v, CEED_REQUEST_IMMEDIATE);
```

A number of function calls in the interface, such as `CeedOperatorApply()`, are intended to support asynchronous execution via their last argument, `CeedRequest*`. The specific (pointer) value used in the above example, `CEED_REQUEST_IMMEDIATE`, is used to express the request (from the user) for the operation to complete before returning from the function call, i.e. to make sure that the result of the operation is available in the output parameters immediately after the call. For a true asynchronous call, one needs to provide the address of a user defined variable. Such a variable can be used later to explicitly wait for the completion of the operation.

3.4 Gallery of QFunctions

LibCEED provides a gallery of built-in *CeedQFunction*s in the `gallery/` directory. The available QFunctions are the ones associated with the mass, the Laplacian, and the identity operators. To illustrate how the user can declare a *CeedQFunction* via the gallery of available QFunctions, consider the selection of the *CeedQFunction* associated with a simple 1D mass matrix (cf. tests/t410-qfunction.c).

```
/// @file
/// Test creation, evaluation, and destruction for QFunction by name
/// \test Test creation, evaluation, and destruction for QFunction by name
#include <ceed.h>
#include <stdio.h>

int main(int argc, char **argv) {
  Ceed          ceed;
  CeedVector    in[16], out[16];
  CeedVector    q_data, dx, w, u, v;
  CeedQFunction qf_setup, qf_mass;
  CeedInt       q = 8;
  CeedScalar    v_true[q];

  CeedInit(argv[1], &ceed);

  CeedVectorCreate(ceed, q, &dx);
  CeedVectorCreate(ceed, q, &w);
  CeedVectorCreate(ceed, q, &u);
  {
    CeedScalar dx_array[q], w_array[q], u_array[q];

    for (CeedInt i = 0; i < q; i++) {
      CeedScalar x = 2. * i / (q - 1) - 1;
      dx_array[i]  = 1;
      w_array[i]   = 1 - x * x;
      u_array[i]   = 2 + 3 * x + 5 * x * x;
      v_true[i]    = w_array[i] * u_array[i];
    }
```

(continues on next page)

(continued from previous page)

```
    CeedVectorSetArray(dx, CEED_MEM_HOST, CEED_COPY_VALUES, dx_array);
    CeedVectorSetArray(w, CEED_MEM_HOST, CEED_COPY_VALUES, w_array);
    CeedVectorSetArray(u, CEED_MEM_HOST, CEED_COPY_VALUES, u_array);
  }
  CeedVectorCreate(ceed, q, &v);
  CeedVectorSetValue(v, 0);
  CeedVectorCreate(ceed, q, &q_data);
  CeedVectorSetValue(q_data, 0);

  CeedQFunctionCreateInteriorByName(ceed, "Mass1DBuild", &qf_setup);
  {
    in[0]  = dx;
    in[1]  = w;
    out[0] = q_data;
    CeedQFunctionApply(qf_setup, q, in, out);
  }

  CeedQFunctionCreateInteriorByName(ceed, "MassApply", &qf_mass);
  {
    in[0]  = w;
    in[1]  = u;
    out[0] = v;
    CeedQFunctionApply(qf_mass, q, in, out);
  }

  // Verify results
  {
    const CeedScalar *v_array;

    CeedVectorGetArrayRead(v, CEED_MEM_HOST, &v_array);
    for (CeedInt i = 0; i < q; i++) {
      if (v_true[i] != v_array[i]) printf("[%" CeedInt_FMT "] v_true %f != v %f\n", i, v_true[i], v_array[i]);
    }
    CeedVectorRestoreArrayRead(v, &v_array);
  }

  CeedVectorDestroy(&dx);
  CeedVectorDestroy(&w);
  CeedVectorDestroy(&u);
  CeedVectorDestroy(&v);
  CeedVectorDestroy(&q_data);
  CeedQFunctionDestroy(&qf_setup);
  CeedQFunctionDestroy(&qf_mass);
  CeedDestroy(&ceed);
  return 0;
}
```

3.5 Interface Principles and Evolution

LibCEED is intended to be extensible via backends that are packaged with the library and packaged separately (possibly as a binary containing proprietary code). Backends are registered by calling

```
    CeedRegister("/cpu/self/ref/serial", CeedInit_Ref, 50);
```

typically in a library initializer or "constructor" that runs automatically. `CeedInit` uses this prefix to find an appropriate backend for the resource.

Source (API) and binary (ABI) stability are important to libCEED. Prior to reaching version 1.0, libCEED does not implement strict semantic versioning across the entire interface. However, user code, including libraries of *CeedQFunction*s, should be source and binary compatible moving from 0.x.y to any later release 0.x.z. We have less experience with external packaging of backends and do not presently guarantee source or binary stability, but we intend to define stability guarantees for libCEED 1.0. We'd love to talk with you if you're interested in packaging backends externally, and will work with you on a practical stability policy.

4 Examples

This section contains a mathematical description of all examples provided with libCEED in the `examples/` directory. These examples are meant to demonstrate use of libCEED from standalone definition of operators to integration with external libraries such as PETSc, MFEM, and Nek5000, as well as more substantial mini-apps.

4.1 Common notation

For most of our examples, the spatial discretization uses high-order finite elements/spectral elements, namely, the high-order Lagrange polynomials defined over P non-uniformly spaced nodes, the Gauss-Legendre-Lobatto (GLL) points, and quadrature points $\{q_i\}_{i=1}^Q$, with corresponding weights $\{w_i\}_{i=1}^Q$ (typically the ones given by Gauss or Gauss-Lobatto quadratures, that are built in the library).

We discretize the domain, $\Omega \subset \mathbb{R}^d$ (with $d = 1, 2, 3$, typically) by letting $\Omega = \bigcup_{e=1}^{N_e} \Omega_e$, with N_e disjoint elements. For most examples we use unstructured meshes for which the elements are hexahedra (although this is not a requirement in libCEED).

The physical coordinates are denoted by $\bm{x} = (x, y, z) \equiv (x_0, x_1, x_2) \in \Omega_e$, while the reference coordinates are represented as $\bm{X} = (X, Y, Z) \equiv (X_0, X_1, X_2) \in \mathrm{I} = [-1, 1]^3$ (for $d = 3$).

4.2 Standalone libCEED

The following two examples have no dependencies, and are designed to be self-contained. For additional examples that use external discretization libraries (MFEM, PETSc, Nek5000 etc.) see the subdirectories in `examples/`.

4.2.1 Ex1-Volume

This example is located in the subdirectory `examples/ceed`. It illustrates a simple usage of libCEED to compute the volume of a given body using a matrix-free application of the mass operator. Arbitrary mesh and solution orders in 1D, 2D, and 3D are supported from the same code.

This example shows how to compute line/surface/volume integrals of a 1D, 2D, or 3D domain Ω respectively, by applying the mass operator to a vector of 1s. It computes:

$$I = \int_{\Omega} 1 \, dV. \tag{4.1}$$

Using the same notation as in *Theoretical Framework*, we write here the vector $u(x) \equiv 1$ in the Galerkin approximation, and find the volume of Ω as

$$\sum_e \int_{\Omega_e} v(x) 1 \, dV \tag{4.2}$$

with $v(x) \in \mathcal{V}_p = \{ v \in H^1(\Omega_e) \,|\, v \in P_p(\boldsymbol{I}), e = 1, \ldots, N_e \}$, the test functions.

4.2.2 Ex2-Surface

This example is located in the subdirectory `examples/ceed`. It computes the surface area of a given body using matrix-free application of a diffusion operator. Similar to *Ex1-Volume*, arbitrary mesh and solution orders in 1D, 2D, and 3D are supported from the same code. It computes:

$$I = \int_{\partial \Omega} 1 \, dS, \tag{4.3}$$

by applying the divergence theorem. In particular, we select $u(\boldsymbol{x}) = x_0 + x_1 + x_2$, for which $\nabla u = [1, 1, 1]^T$, and thus $\nabla u \cdot \hat{\boldsymbol{n}} = 1$.

Given Laplace's equation,

$$\nabla \cdot \nabla u = 0, \text{ for } \boldsymbol{x} \in \Omega,$$

let us multiply by a test function v and integrate by parts to obtain

$$\int_{\Omega} \nabla v \cdot \nabla u \, dV - \int_{\partial \Omega} v \nabla u \cdot \hat{\boldsymbol{n}} \, dS = 0.$$

Since we have chosen u such that $\nabla u \cdot \hat{\boldsymbol{n}} = 1$, the boundary integrand is $v 1 \equiv v$. Hence, similar to (4.2), we can evaluate the surface integral by applying the volumetric Laplacian as follows

$$\int_{\Omega} \nabla v \cdot \nabla u \, dV \approx \sum_e \int_{\partial \Omega_e} v(x) 1 \, dS.$$

4.3 CEED Bakeoff Problems

The Center for Efficient Exascale Discretizations (CEED) uses Bakeoff Problems (BPs) to test and compare the performance of high-order finite element implementations. The definitions of the problems are given on the ceed website. Each of the following bakeoff problems that use external discretization libraries (such as MFEM, PETSc, and Nek5000) are located in the subdirectories `mfem/`, `petsc/`, and `nek5000/`, respectively.

Here we provide a short summary:

User code	Supported BPs
`mfem`	• BP1 (scalar mass operator) with $Q = P + 1$ • BP3 (scalar Laplace operator) with $Q = P + 1$
`petsc`	• BP1 (scalar mass operator) with $Q = P + 1$ • BP2 (vector mass operator) with $Q = P + 1$ • BP3 (scalar Laplace operator) with $Q = P + 1$ • BP4 (vector Laplace operator) with $Q = P + 1$ • BP5 (collocated scalar Laplace operator) with $Q = P$ • BP6 (collocated vector Laplace operator) with $Q = P$
`nek5000`	• BP1 (scalar mass operator) with $Q = P + 1$ • BP3 (scalar Laplace operator) with $Q = P + 1$

These are all **T-vector**-to-**T-vector** and include parallel scatter, element scatter, element evaluation kernel, element gather, and parallel gather (with the parallel gathers/scatters done externally to libCEED).

BP1 and BP2 are L^2 projections, and thus have no boundary condition. The rest of the BPs have homogeneous Dirichlet boundary conditions.

The BPs are parametrized by the number P of Gauss-Legendre-Lobatto nodal points (with $P = p + 1$, and p the degree of the basis polynomial) for the Lagrange polynomials, as well as the number of quadrature points, Q. A Q-point Gauss-Legendre quadrature is used for all BPs except BP5 and BP6, which choose $Q = P$ and Gauss-Legendre-Lobatto quadrature to collocate with the interpolation nodes. This latter choice is popular in applications that use spectral element methods because it produces a diagonal mass matrix (enabling easy explicit time integration) and significantly reduces the number of floating point operations to apply the operator.

4.3.1 Mass Operator

The Mass Operator used in BP1 and BP2 is defined via the L^2 projection problem, posed as a weak form on a Hilbert space $V^p \subset H^1$, i.e., find $u \in V^p$ such that for all $v \in V^p$

$$\langle v, u\rangle = \langle v, f\rangle, \tag{4.4}$$

where $\langle v, u\rangle$ and $\langle v, f\rangle$ express the continuous bilinear and linear forms, respectively, defined on V^p, and, for sufficiently regular u, v, and f, we have:

$$\begin{aligned}\langle v, u\rangle &:= \int_\Omega v\, u\, dV,\\ \langle v, f\rangle &:= \int_\Omega v f dV.\end{aligned}$$

Following the standard finite/spectral element approach, we formally expand all functions in terms of basis functions, such as

$$\begin{aligned} u(\boldsymbol{x}) &= \sum_{j=1}^{n} u_j\, \phi_j(\boldsymbol{x}),\\ v(\boldsymbol{x}) &= \sum_{i=1}^{n} v_i\, \phi_i(\boldsymbol{x}).\end{aligned} \tag{4.5}$$

The coefficients $\{u_j\}$ and $\{v_i\}$ are the nodal values of u and v, respectively. Inserting the expressions (4.5) into (4.4), we obtain the inner-products

$$\langle v, u\rangle = \boldsymbol{v}^T M \boldsymbol{u}, \qquad \langle v, f\rangle = \boldsymbol{v}^T \boldsymbol{b}\,. \tag{4.6}$$

Here, we have introduced the mass matrix, M, and the right-hand side, $\boldsymbol{b}$,

$$M_{ij} := (\phi_i, \phi_j), \qquad b_i := \langle \phi_i, f\rangle,$$

each defined for index sets $i, j \in \{1, \dots, n\}$.

4.3.2 Laplace's Operator

The Laplace's operator used in BP3-BP6 is defined via the following variational formulation, i.e., find $u \in V^p$ such that for all $v \in V^p$

$$a(v, u) = \langle v, f\rangle,$$

where now $a(v, u)$ expresses the continuous bilinear form defined on V^p for sufficiently regular u, v, and f, that is:

$$\begin{aligned} a(v, u) &:= \int_\Omega \nabla v \cdot \nabla u \, dV, \\ \langle v, f\rangle &:= \int_\Omega v f \, dV. \end{aligned}$$

After substituting the same formulations provided in (4.5), we obtain

$$a(v, u) = \boldsymbol{v}^T K \boldsymbol{u},$$

in which we have introduced the stiffness (diffusion) matrix, K, defined as

$$K_{ij} = a(\phi_i, \phi_j),$$

for index sets $i, j \in \{1, \dots, n\}$.

4.4 PETSc demos and BPs

4.4.1 Area

This example is located in the subdirectory `examples/petsc`. It demonstrates a simple usage of libCEED with PETSc to calculate the surface area of a closed surface. The code uses higher level communication protocols for mesh handling in PETSc's DMPlex. This example has the same mathematical formulation as *Ex1-Volume*, with the exception that the physical coordinates for this problem are $\boldsymbol{x} = (x, y, z) \in \mathbb{R}^3$, while the coordinates of the reference element are $\boldsymbol{X} = (X, Y) \equiv (X_0, X_1) \in \mathrm{I} = [-1, 1]^2$.

4.4.1.1 Cube

This is one of the test cases of the computation of the *Area* of a 2D manifold embedded in 3D. This problem can be run with:

```
./area -problem cube
```

This example uses the following coordinate transformations for the computation of the geometric factors: from the physical coordinates on the cube, denoted by $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y}, \bar{z})$, and physical coordinates on the discrete surface, denoted by $\boldsymbol{x} = (x, y)$, to $\boldsymbol{X} = (X, Y) \in \mathrm{I}$ on the reference element, via the chain rule

$$\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}}_{(2\times2)} = \frac{\partial \boldsymbol{x}}{\partial \bar{\boldsymbol{x}}}_{(2\times3)} \frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}}_{(3\times2)}, \tag{4.7}$$

with Jacobian determinant given by

$$|J| = \left\| col_1 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right\| \left\| col_2 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right\| \tag{4.8}$$

We note that in equation (4.7), the right-most Jacobian matrix $\partial \bar{\boldsymbol{x}} / \partial \boldsymbol{X}_{(3\times2)}$ is provided by the library, while $\partial \boldsymbol{x} / \partial \bar{\boldsymbol{x}}_{(2\times3)}$ is provided by the user as

$$\left[col_1 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) / \left\| col_1 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right\|, col_2 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) / \left\| col_2 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right\| \right]^T_{(2\times3)}.$$

4.4.1.2 Sphere

This problem computes the surface *Area* of a tensor-product discrete sphere, obtained by projecting a cube inscribed in a sphere onto the surface of the sphere. This discrete surface is sometimes referred to as a cubed-sphere (an example of such as a surface is given in figure Fig. 4.1). This problem can be run with:

```
./area -problem sphere
```

This example uses the following coordinate transformations for the computation of the geometric factors: from the physical coordinates on the sphere, denoted by $\mathring{\boldsymbol{x}} = (\mathring{x}, \mathring{y}, \mathring{z})$, and physical coordinates on the discrete surface, denoted by $\boldsymbol{x} = (x, y, z)$ (depicted, for simplicity, as coordinates on a circle and 1D linear element in figure Fig. 4.2), to $\boldsymbol{X} = (X, Y) \in \mathrm{I}$ on the reference element, via the chain rule

$$\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}_{(3\times2)} = \frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{x}}_{(3\times3)} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}}_{(3\times2)}, \tag{4.9}$$

with Jacobian determinant given by

$$|J| = \left| col_1 \left(\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \times col_2 \left(\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right|. \tag{4.10}$$

We note that in equation (4.9), the right-most Jacobian matrix $\partial \boldsymbol{x} / \partial \boldsymbol{X}_{(3\times2)}$ is provided by the library, while $\partial \mathring{\boldsymbol{x}} / \partial \boldsymbol{x}_{(3\times3)}$ is provided by the user with analytical derivatives. In particular, for a sphere of radius 1, we have

$$\mathring{\boldsymbol{x}}(\boldsymbol{x}) = \frac{1}{\|\boldsymbol{x}\|} \boldsymbol{x}_{(3\times1)}$$

and thus

$$\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{x}} = \frac{1}{\|\boldsymbol{x}\|} \boldsymbol{I}_{(3\times3)} - \frac{1}{\|\boldsymbol{x}\|^3} (\boldsymbol{x}\boldsymbol{x}^T)_{(3\times3)}.$$

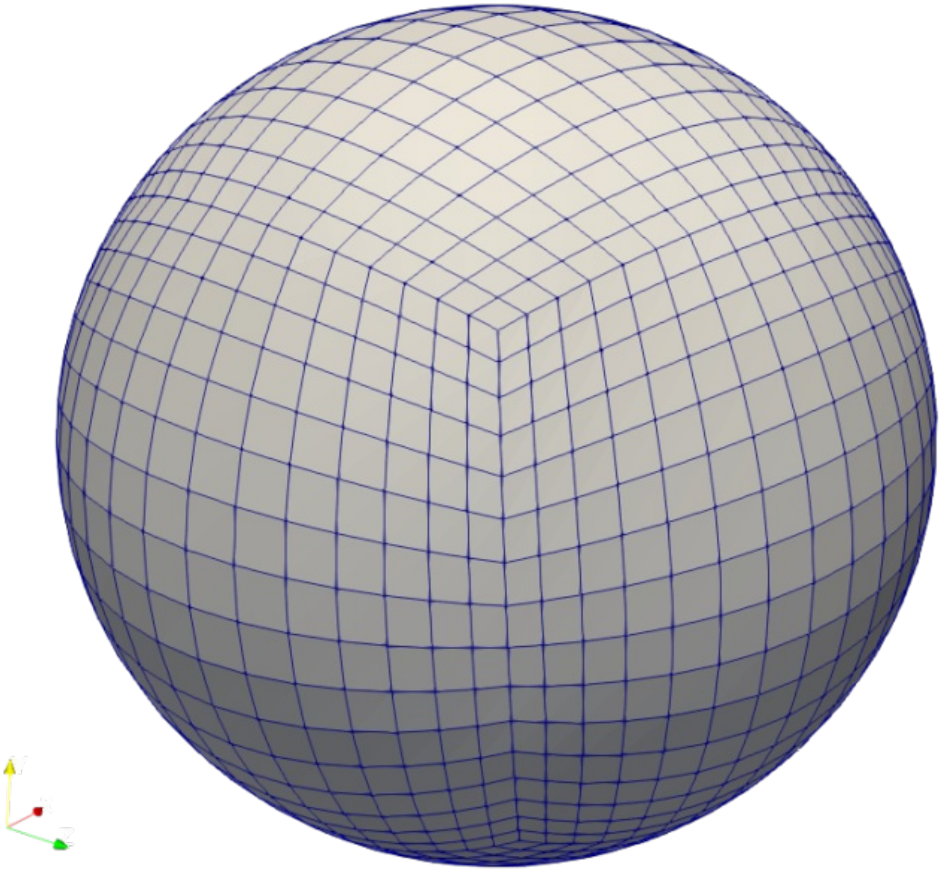


Fig. 4.1: Example of a cubed-sphere, i.e., a tensor-product discrete sphere, obtained by projecting a cube inscribed in a sphere onto the surface of the sphere.

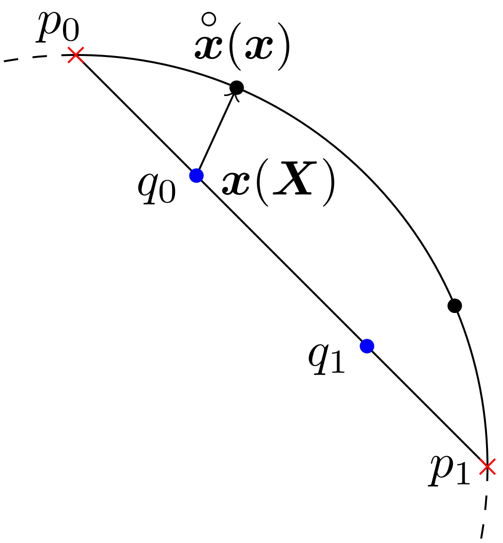


Fig. 4.2: Sketch of coordinates mapping between a 1D linear element and a circle. In the case of a linear element the two nodes, p_0 and p_1, marked by red crosses, coincide with the endpoints of the element. Two quadrature points, q_0 and q_1, marked by blue dots, with physical coordinates denoted by $\boldsymbol{x}(\boldsymbol{X})$, are mapped to their corresponding radial projections on the circle, which have coordinates $\mathring{\boldsymbol{x}}(\boldsymbol{x})$.

4.4.2 Bakeoff problems and generalizations

The PETSc examples in this directory include a full suite of parallel *bakeoff problems* (BPs) using a "raw" parallel decomposition (see `bpsraw.c`) and using PETSc's `DMPlex` for unstructured grid management (see `bps.c`). A generalization of these BPs to the surface of the cubed-sphere are available in `bpssphere.c`.

4.4.2.1 Bakeoff problems on the cubed-sphere

For the L^2 projection problems, BP1-BP2, that use the mass operator, the coordinate transformations and the corresponding Jacobian determinant, equation (4.10), are the same as in the *Sphere* example. For the Poisson's problem, BP3-BP6, on the cubed-sphere, in addition to equation (4.10), the pseudo-inverse of $\partial\mathring{\boldsymbol{x}}/\partial\boldsymbol{X}$ is used to derive the contravariant metric tensor (please see figure Fig. 4.2 for a reference of the notation used). We begin by expressing the Moore-Penrose (left) pseudo-inverse:

$$\frac{\partial \boldsymbol{X}}{\partial \mathring{\boldsymbol{x}}}_{(2\times3)} \equiv \left(\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}\right)^{+}_{(2\times3)} = \left(\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}^T_{(2\times3)} \frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}_{(3\times2)}\right)^{-1} \frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}^T_{(2\times3)}. \tag{4.11}$$

This enables computation of gradients of an arbitrary function $u(\mathring{\boldsymbol{x}})$ in the embedding space as

$$\frac{\partial u}{\partial \mathring{\boldsymbol{x}}}_{(1\times3)} = \frac{\partial u}{\partial \boldsymbol{X}}_{(1\times2)} \frac{\partial \boldsymbol{X}}{\partial \mathring{\boldsymbol{x}}}_{(2\times3)}$$

and thus the weak Laplacian may be expressed as

$$\int_\Omega \frac{\partial v}{\partial \mathring{\boldsymbol{x}}} \left(\frac{\partial u}{\partial \mathring{\boldsymbol{x}}}\right)^T dS = \int_\Omega \frac{\partial v}{\partial \boldsymbol{X}} \underbrace{\frac{\partial \boldsymbol{X}}{\partial \mathring{\boldsymbol{x}}} \left(\frac{\partial \boldsymbol{X}}{\partial \mathring{\boldsymbol{x}}}\right)^T}_{\boldsymbol{g}_{(2\times2)}} \left(\frac{\partial u}{\partial \boldsymbol{X}}\right)^T dS \tag{4.12}$$

where we have identified the 2×2 contravariant metric tensor $\boldsymbol{g}$ (sometimes written $\boldsymbol{g}^{ij}$), and where now Ω represents the surface of the sphere, which is a two-dimensional closed surface embedded in the three-dimensional Euclidean space $\mathbb{R}^3$. This expression can be simplified to avoid the explicit Moore-Penrose pseudo-inverse,

$$g = \left(\frac{\partial \mathring{x}}{\partial X}^T \frac{\partial \mathring{x}}{\partial X} \right)^{-1}_{(2\times 2)} \frac{\partial \mathring{x}}{\partial X}^T_{(2\times 3)} \frac{\partial \mathring{x}}{\partial X}_{(3\times 2)} \left(\frac{\partial \mathring{x}}{\partial X}^T \frac{\partial \mathring{x}}{\partial X} \right)^{-T}_{(2\times 2)} = \left(\frac{\partial \mathring{x}}{\partial X}^T \frac{\partial \mathring{x}}{\partial X} \right)^{-1}_{(2\times 2)}$$

where we have dropped the transpose due to symmetry. This allows us to simplify (4.12) as

$$\int_\Omega \frac{\partial v}{\partial \mathring{x}} \left(\frac{\partial u}{\partial \mathring{x}} \right)^T dS = \int_\Omega \frac{\partial v}{\partial X} \underbrace{\left(\frac{\partial \mathring{x}}{\partial X}^T \frac{\partial \mathring{x}}{\partial X} \right)^{-1}}_{g_{(2\times 2)}} \left(\frac{\partial u}{\partial X} \right)^T dS,$$

which is the form implemented in `qfunctions/bps/bp3sphere.h`.

4.4.3 Multigrid

This example is located in the subdirectory `examples/petsc`. It investigates p-multigrid for the Poisson problem, equation (4.13), using an unstructured high-order finite element discretization. All of the operators associated with the geometric multigrid are implemented in libCEED.

$$-\nabla \cdot (\kappa(x) \nabla x) = g(x) \tag{4.13}$$

The Poisson operator can be specified with the decomposition given by the equation in figure *Operator Decomposition*, and the restriction and prolongation operators given by interpolation basis operations, $\boldsymbol{B}$, and $\boldsymbol{B}^T$, respectively, act on the different grid levels with corresponding element restrictions, $\boldsymbol{G}$. These three operations can be exploited by existing matrix-free multigrid software and smoothers. Preconditioning based on the libCEED finite element operator decomposition is an ongoing area of research.

4.5 Compressible Navier-Stokes mini-app

This example is located in the subdirectory `examples/fluids`. It solves the time-dependent Navier-Stokes equations of compressible gas dynamics in a static Eulerian three-dimensional frame using unstructured high-order finite/spectral element spatial discretizations and explicit or implicit high-order time-stepping (available in PETSc). Moreover, the Navier-Stokes example has been developed using PETSc, so that the pointwise physics (defined at quadrature points) is separated from the parallelization and meshing concerns.

4.5.1 Running the mini-app

The Navier-Stokes mini-app is controlled via command-line options. The following options are common among all problem types:

Option	Description
`-ceed`	CEED resource specifier
`-test_type`	Run in test mode and specify whether solut
`-compare_final_state_atol`	Test absolute tolerance
`-compare_final_state_filename`	Test filename

Tabl

Option	Description
`-problem`	Problem to solve (`advection`, `advectior`
`-implicit`	Use implicit time integrator formulation
`-degree`	Polynomial degree of tensor product basis (
`-q_extra`	Number of extra quadrature points
`-ts_monitor_solution`	PETSc output format, such as `cgns:outpu`
`-ts_monitor_solution_interval`	Number of time steps between visualizatior
`-viewer_cgns_batch_size`	Number of frames written per CGNS file if
`-checkpoint_interval`	Number of steps between writing binary ch
`-checkpoint_vtk`	Checkpoints include VTK (`*.vtu`) files for
`-viz_refine`	Use regular refinement for VTK visualizatic
`-output_dir`	Output directory for binary checkpoints anc
`-output_add_stepnum2bin`	Whether to add step numbers to output bin
`-continue`	Continue from previous solution (input is s
`-continue_filename`	Path to solution binary file from which to cc
`-continue_time_filename`	Path to time stamp binary file (only for lega
`-bc_wall`	Use wall boundary conditions on this list of
`-wall_comps`	An array of constrained component number
`-bc_slip_x`	Use slip boundary conditions, for the x com
`-bc_slip_y`	Use slip boundary conditions, for the y com
`-bc_slip_z`	Use slip boundary conditions, for the z com
`-bc_inflow`	Use inflow boundary conditions on this list
`-bc_outflow`	Use outflow boundary conditions on this lis
`-bc_freestream`	Use freestream boundary conditions on this
`-ts_monitor_turbulence_spanstats_collect_interval`	Number of timesteps between statistics colle
`-ts_monitor_turbulence_spanstats_viewer`	Sets the PetscViewer for the statistics file wr
`-ts_monitor_turbulence_spanstats_viewer_interval`	Number of timesteps between statistics file
`-ts_monitor_turbulence_spanstats_viewer_cgns_batch_size`	Number of frames written per CGNS file if
`-ts_monitor_wall_force`	Viewer for the force on each no-slip wall, e.g
`-mesh_transform`	Transform the mesh, usually for an initial bo
`-snes_view`	View PETSc `SNES` nonlinear solver configur
`-log_view`	View PETSc performance log
`-help`	View comprehensive information about rur

For the case of a square/cubic mesh, the list of face indices to be used with `-bc_wall`, `bc_inflow`, `bc_outflow`, `bc_freestream` and/or `-bc_slip_x`, `-bc_slip_y`, and `-bc_slip_z` are:

Table 4.2: 2D Face ID Labels

PETSc Face Name	Cartesian direction	Face ID
faceMarkerBottom	-z	1
faceMarkerRight	+x	2
faceMarkerTop	+z	3
faceMarkerLeft	-x	4

Table 4.3: 3D Face ID Labels

PETSc Face Name	Cartesian direction	Face ID
faceMarkerBottom	-z	1
faceMarkerTop	+z	2
faceMarkerFront	-y	3
faceMarkerBack	+y	4
faceMarkerRight	+x	5
faceMarkerLeft	-x	6

4.5.1.1 Boundary conditions

Boundary conditions for compressible viscous flows are notoriously tricky. Here we offer some recommendations

Inflow

If in a region where the flow velocity is known (e.g., away from viscous walls), use `bc_freestream`, which solves a Riemann problem and can handle inflow and outflow (simultaneously and dynamically). It is stable and the least reflective boundary condition for acoustics.

If near a viscous wall, you may want a specified inflow profile. Use `bc_inflow` and see *Blasius boundary layer* and discussion of synthetic turbulence generation for ways to analytically generate developed inflow profiles. These conditions may be either weak or strong, with the latter specifying velocity and temperature as essential boundary conditions and evaluating a boundary integral for the mass flux. The strong approach gives sharper resolution of velocity structures. We have described the primitive variable formulation here; the conservative variants are similar, but not equivalent.

Outflow

If you know the complete exterior state, `bc_freestream` is the least reflective boundary condition, but is disruptive to viscous flow structures. If thermal anomalies must exit the domain, the Riemann solver must resolve the contact wave to avoid reflections. The default Riemann solver, HLLC, is sufficient in this regard while the simpler HLL converts thermal structures exiting the domain into grid-scale reflecting acoustics.

If acoustic reflections are not a concern and/or the flow is impacted by walls or interior structures that you wish to resolve to near the boundary, choose `bc_outflow`. This condition (with default `outflow_type: riemann`) is stable for both inflow and outflow, so can be used in areas that have recirculation and lateral boundaries in which the flow fluctuates.

The simpler `bc_outflow` variant, `outflow_type: pressure`, requires that the flow be a strict outflow (or the problem becomes ill-posed and the solver will diverge). In our experience, `riemann` is slightly less reflective but produces similar flows in cases of strict outflow. The `pressure` variant is retained to facilitate comparison with other codes, such as PHASTA-C, but we recommend `riemann` for general use.

Periodicity

PETSc provides two ways to specify periodicity:

1. Topological periodicity, in which the donor and receiver dofs are the same, obtained using:

```
dm_plex:
  shape: box
  box_faces: 10,12,4
  box_bd: none,none,periodic
```

The coordinates for such cases are stored as a new field with special cell-based indexing to enable wrapping through the boundary. This choice of coordinates prevents evaluating boundary integrals that cross the periodicity, such as for the outflow Riemann problem in the presence of spanwise periodicity.

2. Isoperiodicity, in which the donor and receiver dofs are distinct in local vectors. This is obtained using `zbox`, as in:

```
dm_plex:
  shape: zbox
  box_faces: 10,12,4
  box_bd: none,none,periodic
```

Isoperiodicity enables standard boundary integrals, and is recommended for general use. At the time of this writing, it only supports one direction of periodicity. The `zbox` method uses Z-ordering to construct the mesh in parallel and provide an adequate initial partition, which makes it higher performance and avoids needing a partitioning package.

4.5.1.2 Advection

For testing purposes, there is a reduced mode for pure advection, which holds density ρ and momentum density $\rho \boldsymbol{u}$ constant while advecting "total energy density" E. These are available in 2D and 3D.

2D advection

For the 2D advection problem, the following additional command-line options are available:

Table 4.4: Advection2D Runtime Options

Option	Description	Default value	Unit
`-rc`	Characteristic radius of thermal bubble	`1000`	`m`
`-units_meter`	1 meter in scaled length units	`1E-2`	
`-units_second`	1 second in scaled time units	`1E-2`	
`-units_kilogram`	1 kilogram in scaled mass units	`1E-6`	
`-strong_form`	Strong (1) or weak/integrated by parts (0) residual	`0`	
`-stab`	Stabilization method (`none`, `su`, or `supg`)	`none`	
`-CtauS`	Scale coefficient for stabilization tau (nondimensional)	`0`	
`-wind_type`	Wind type in Advection (`rotation` or `translation`)	`rotation`	
`-wind_translation`	Constant wind vector when `-wind_type translation`	`1,0,0`	
`-E_wind`	Total energy of inflow wind when `-wind_type translation`	`1E6`	`J`

An example of the `rotation` mode can be run with:

```
./navierstokes -problem advection2d -dm_plex_box_faces 20,20 -dm_plex_box_lower 0,0 -
↪dm_plex_box_upper 1000,1000 -bc_wall 1,2,3,4 -wall_comps 4 -wind_type rotation -
↪implicit -stab supg
```

and the `translation` mode with:

```
./navierstokes -problem advection2d -dm_plex_box_faces 20,20 -dm_plex_box_lower 0,0 -
↪dm_plex_box_upper 1000,1000 -units_meter 1e-4 -wind_type translation -wind_
↪translation 1,-.5 -bc_inflow 1,2,3,4
```

Note the lengths in `-dm_plex_box_upper` are given in meters, and will be nondimensionalized according to `-units_meter`.

3D advection

For the 3D advection problem, the following additional command-line options are available:

Table 4.5: Advection3D Runtime Options

Option	Description	Default value	Unit
`-rc`	Characteristic radius of thermal bubble	`1000`	`m`
`-units_meter`	1 meter in scaled length units	`1E-2`	
`-units_second`	1 second in scaled time units	`1E-2`	
`-units_kilogram`	1 kilogram in scaled mass units	`1E-6`	
`-strong_form`	Strong (1) or weak/integrated by parts (0) residual	`0`	
`-stab`	Stabilization method (`none`, `su`, or `supg`)	`none`	
`-CtauS`	Scale coefficient for stabilization tau (nondimensional)	`0`	
`-wind_type`	Wind type in Advection (`rotation` or `translation`)	`rotation`	
`-wind_translation`	Constant wind vector when `-wind_type translation`	`1,0,0`	
`-E_wind`	Total energy of inflow wind when `-wind_type translation`	`1E6`	`J`
`-bubble_type`	`sphere` (3D) or `cylinder` (2D)	`sphere`	
`-bubble_continuity`	`smooth`, `back_sharp`, or `thick`	`smooth`	

An example of the `rotation` mode can be run with:

```
./navierstokes -problem advection -dm_plex_box_faces 10,10,10 -dm_plex_dim 3 -dm_plex_
↪box_lower 0,0,0 -dm_plex_box_upper 8000,8000,8000 -bc_wall 1,2,3,4,5,6 -wall_comps␣
↪4 -wind_type rotation -implicit -stab su
```

and the `translation` mode with:

```
./navierstokes -problem advection -dm_plex_box_faces 10,10,10 -dm_plex_dim 3 -dm_plex_
↪box_lower 0,0,0 -dm_plex_box_upper 8000,8000,8000 -wind_type translation -wind_
↪translation .5,-1,0 -bc_inflow 1,2,3,4,5,6
```

4.5.1.3 Inviscid Ideal Gas

Isentropic Euler vortex

For the Isentropic Vortex problem, the following additional command-line options are available:

Table 4.6: Isentropic Vortex Runtime Options

Option	Description	Default value	Unit
`-center`	Location of vortex center	`(lx,ly,lz)/2`	`(m,m,m)`
`-units_meter`	1 meter in scaled length units	`1E-2`	
`-units_second`	1 second in scaled time units	`1E-2`	
`-mean_velocity`	Background velocity vector	`(1,1,0)`	
`-vortex_strength`	Strength of vortex < 10	`5`	
`-c_tau`	Stabilization constant	`0.5`	

This problem can be run with:

```
./navierstokes -problem euler_vortex -dm_plex_box_faces 20,20,1 -dm_plex_box_lower 0,
↪0,0 -dm_plex_box_upper 1000,1000,50 -dm_plex_dim 3 -bc_inflow 4,6 -bc_outflow 3,5 -
↪bc_slip_z 1,2 -mean_velocity .5,-.8,0.
```

Sod shock tube

For the Shock Tube problem, the following additional command-line options are available:

Table 4.7: Shock Tube Runtime Options

Option	Description	Default value	Unit
`-units_meter`	1 meter in scaled length units	`1E-2`	
`-units_second`	1 second in scaled time units	`1E-2`	
`-yzb`	Use YZB discontinuity capturing	`none`	
`-stab`	Stabilization method (`none`, `su`, or `supg`)	`none`	

This problem can be run with:

```
./navierstokes -problem shocktube -yzb -stab su -bc_slip_z 3,4 -bc_slip_y 1,2 -bc_
↪wall 5,6 -dm_plex_dim 3 -dm_plex_box_lower 0,0,0 -dm_plex_box_upper 1000,100,100 -
↪dm_plex_box_faces 200,1,1 -units_second 0.1
```

4.5.1.4 Newtonian viscosity, Ideal Gas

For the Density Current, Channel, and Blasius problems, the following common command-line options are available:

Table 4.8: Newtonian Ideal Gas problems Runtime

Option	Description
`-units_meter`	1 meter in scaled length units
`-units_second`	1 second in scaled time units

Table 4.8 – continued from previous pa

Option	Description
-units_kilogram	1 kilogram in scaled mass units
-units_Kelvin	1 Kelvin in scaled temperature units
-stab	Stabilization method (none, su, or supg)
-c_tau	Stabilization constant, c_τ
-Ctau_t	Stabilization time constant, C_t
-Ctau_v	Stabilization viscous constant, C_v
-Ctau_C	Stabilization continuity constant, C_c
-Ctau_M	Stabilization momentum constant, C_m
-Ctau_E	Stabilization energy constant, C_E
-cv	Heat capacity at constant volume
-cp	Heat capacity at constant pressure
-gravity	Gravitational acceleration vector
-lambda	Stokes hypothesis second viscosity coefficient
-mu	Shear dynamic viscosity coefficient
-k	Thermal conductivity
-newtonian_unit_tests	Developer option to test properties
-state_var	State variables to solve solution with. conservative $(\rho, \rho\boldsymbol{u}, \rho e)$ or prin
-idl_decay_time	Characteristic timescale of the pressure deviance decay. The timestep is g
-idl_start	Start of IDL in the x direction
-idl_length	Length of IDL in the positive x direction
-sgs_model_type	Type of subgrid stress model to use. Currently only data_driven is ava
-sgs_model_dd_leakyrelu_alpha	Slope parameter for Leaky ReLU activation function. 0 corresponds to no
-sgs_model_dd_parameter_dir	Path to directory with data-driven model parameters (weights, biases, et
-diff_filter_monitor	Enable differential filter TSMonitor
-diff_filter_grid_based_width	Use filter width based on the grid size
-diff_filter_width_scaling	Anisotropic scaling for filter width in wall-aligned coordinates (snz)
-diff_filter_kernel_scaling	Scaling to make differential kernel size equivalent to other filter kernels
-diff_filter_wall_damping_function	Damping function to use at the wall for anisotropic filtering (none, van_
-diff_filter_wall_damping_constant	Constant for the wall-damping function. A^+ for van_driest damping f
-diff_filter_friction_length	Friction length associated with the flow, δ_ν. Used in wall-damping functi

Gaussian Wave

The Gaussian wave problem has the following command-line options in addition to the Newtonian Ideal Gas options:

Table 4.9: Gaussian Wave Runtime Options

Option	Description	Default value	Unit
-freestream_riemann	Riemann solver for boundaries (HLL or HLLC)	hllc	
-freestream_velocity	Freestream velocity vector	0,0,0	m/s
-freestream_temperature	Freestream temperature	288	K
-freestream_pressure	Freestream pressure	1.01e5	Pa
-epicenter	Coordinates of center of perturbation	0,0,0	m
-amplitude	Amplitude of the perturbation	0.1	
-width	Width parameter of the perturbation	0.002	m

This problem can be run with the `gaussianwave.yaml` file via:

```
./navierstokes -options_file gaussianwave.yaml
```

```yaml
problem: gaussian_wave
mu: 0 # Effectively solving Euler momentum equations

dm_plex_box_faces: 40,40,1
dm_plex_box_upper: 1,1,0.025
dm_plex_box_lower: 0,0,0
dm_plex_dim: 3
bc_freestream: 4,6,3,5
bc_slip_z: 1,2

reference:
  temperature: 0.25
  pressure: 71.75
freestream:
  # riemann: hll # causes thermal bubble to reflect acoustic waves from boundary
  velocity: 2,2,0

epicenter: 0.33,0.75,0
amplitude: 2
width: 0.05

ts:
  adapt_type: none
  max_steps: 100
  dt: 2e-3
  type: alpha
  alpha_radius: 0.5
  #monitor_solution: cgns:nwave.cgns
  #monitor_solution_interval: 10

implicit: true
stab: supg
state_var: primitive

snes_rtol: 1e-4
ksp_rtol: 1e-2
snes_lag_jacobian: 20
snes_lag_jacobian_persists:

## Demonstrate acoustic wave dissipation using an internal damping layer
# idl:
#   decay_time: 2e-3
#   start: 0
#   length: .25
```

Vortex Shedding - Flow past Cylinder

The vortex shedding, flow past cylinder problem has the following command-line options in addition to the Newtonian Ideal Gas options:

Table 4.10: Vortex Shedding Runtime Options

Option	Description	Default value	Unit
`-freestream_velocity`	Freestream velocity vector	`0,0,0`	`m/s`
`-freestream_temperature`	Freestream temperature	`288`	`K`
`-freestream_pressure`	Freestream pressure	`1.01e5`	`Pa`

The initial condition is taken from `-reference_temperature` and `-reference_pressure`. To run this problem, first generate a mesh:

```
$ make -C examples/fluids/meshes
```

Then run by building the executable and running:

```
$ make build/fluids-navierstokes
$ mpiexec -n 6 build/fluids-navierstokes -options_file examples/fluids/vortexshedding.yaml -{ts,snes}_monitor_
```

The vortex shedding period is roughly 5.6 and this problem runs until time 100 (2000 time steps). The above run writes a file named `force.csv` (see `ts_monitor_wall_force` in `vortexshedding.yaml`), which can be postprocessed by running to create a figure showing lift and drag coefficients over time.

```
$ python examples/fluids/postprocess/vortexshedding.py
```

```yaml
problem: newtonian

# Time Stepping Settings
implicit: true
stab: supg

checkpoint_interval: 10

ts:
  adapt_type: 'none'
  type: alpha
  dt: .05
  max_time: 100
  alpha_radius: 0.5
  monitor_solution: cgns:vortexshedding-q3-g1-n08.cgns
  monitor_solution_interval: 5
  monitor_wall_force: ascii:force.csv:ascii_csv

# Reference state is used for the initial condition, zero velocity by default.

# This choice of pressure and temperature have a density of 1 and acoustic speed
# of 100. With velocity 1, this flow is Mach 0.01.
reference:
  pressure: 7143
  temperature: 24.92

# If the the outflow is placed close to the cylinder, this will recirculate cold
```

(continues on next page)

(continued from previous page)

```yaml
# fluid, demonstrating how the outflow BC is stable despite recirculation.
outflow:
  temperature: 20

# Freestream inherits reference state as default
freestream:
  velocity: 1,0,0
# Small gravity vector to break symmetry so shedding can start
g: 0,-.01,0

# viscosity corresponds to Reynolds number 100
mu: 0.01
k: 14.34 # thermal conductivity, Pr = 0.71 typical of air

## DM Settings:
degree: 3
dm_plex_filename: examples/fluids/meshes/cylinder-q1-n08.msh

# Boundary Settings
bc_slip_z: 6
bc_wall: 5
bc_freestream: 1
bc_outflow: 2
bc_slip_y: 3,4
wall_comps: 1,2,3

# Primitive variables are preferred at low Mach number
state_var: primitive

dm_view:
ts_monitor:
snes_lag_jacobian: 20
snes_lag_jacobian_persists:

#pmat_pbdiagonal:
#ksp_type: bcgsl
#pc_type: vpbjacobi
amat_type: shell
```

Density current

The Density Current problem has the following command-line options in addition to the Newtonian Ideal Gas options:

Table 4.11: Density Current Runtime Options

Option	Description	Default value	Unit
-center	Location of bubble center	(lx,ly, lz)/2	(m,m, m)
-dc_axis	Axis of density current cylindrical anomaly, or (0,0,0) for spherically symmetric	(0,0,0)	
-rc	Characteristic radius of thermal bubble	1000	m
-theta0	Reference potential temperature	300	K
-thetaC	Perturbation of potential temperature	-15	K
-P0	Atmospheric pressure	1E5	Pa
-N	Brunt-Vaisala frequency	0.01	1/s

This problem can be run with:

```
./navierstokes -problem density_current -dm_plex_box_faces 16,1,8 -degree 1 -dm_plex_
↪box_lower 0,0,0 -dm_plex_box_upper 2000,125,1000 -dm_plex_dim 3 -rc 400. -bc_wall 1,
↪2,5,6 -wall_comps 1,2,3 -bc_slip_y 3,4 -mu 75
```

Channel flow

The Channel problem has the following command-line options in addition to the Newtonian Ideal Gas options:

Table 4.12: Channel Runtime Options

Option	Description	Default value	Unit
-umax	Maximum/centerline velocity of the flow	10	m/s
-theta0	Reference potential temperature	300	K
-P0	Atmospheric pressure	1E5	Pa
-body_force_scale	Multiplier for body force (-1 for flow reversal)	1	

This problem can be run with the `channel.yaml` file via:

```
./navierstokes -options_file channel.yaml
```

```yaml
problem: 'channel'
mu: .01

umax: 40
implicit: true
ts:
  type: 'beuler'
  adapt_type: 'none'
  dt: 5e-6

q_extra: 2

dm_plex_box_lower: 0,0,0
dm_plex_box_upper: .01,.01,.001
dm_plex_dim: 3
degree: 1
```

(continues on next page)

(continued from previous page)

```
dm_plex_box_faces: 10,10,1
bc_slip_z: 1,2
bc_wall: 3,4
wall_comps: 1,2,3
dm_plex_box_bd: 'periodic,none,none'
```

Blasius boundary layer

The Blasius problem has the following command-line options in addition to the Newtonian Ideal Gas options:

Table 4.13: Blasius Runtime Options

Option	Description	Default value	Unit
`-velocity_infinity`	Freestream velocity	`40`	`m/s`
`-temperature_infinity`	Freestream temperature	`288`	`K`
`-temperature_wall`	Wall temperature	`288`	`K`
`-delta0`	Boundary layer height at the inflow	`4.2e-3`	`m`
`-P0`	Atmospheric pressure	`1.01E5`	`Pa`
`-platemesh_modify_mesh`	Whether to modify the mesh using the given options below.	`false`	
`-platemesh_refine_height`	Height at which `-platemesh_Ndelta` number of elements should refined into	`5.9E-4`	`m`
`-platemesh_Ndelta`	Number of elements to keep below `-platemesh_refine_height`	`45`	
`-platemesh_growth`	Growth rate of the elements in the refinement region	`1.08`	
`-platemesh_top_angle`	Downward angle of the top face of the domain. This face serves as an outlet.	`5`	`degrees`
`-platemesh_y_node_locs_path`	Path to file with y node locations. If empty, will use mesh warping instead.	`""`	
`-stg_use`	Whether to use STG for the inflow conditions	`false`	
`-n_chebyshev`	Number of Chebyshev terms	`20`	
`-chebyshev_`	Prefix for Chebyshev snes solve		

This problem can be run with the `blasius.yaml` file via:

```
./navierstokes -options_file blasius.yaml
```

```
problem: 'blasius'

implicit: true
ts:
  adapt_type: 'none'
  type: 'beuler'
  dt: 2e-6
  max_time: 1.0e-3
  #monitor_solution: cgns:blasius-%d.cgns
  #monitor_solution_interval: 10
checkpoint_interval: 10
```

(continues on next page)

(continued from previous page)

```
## Linear Settings:
degree: 1
dm_plex_box_faces: 40,60,1
mesh_transform: platemesh
platemesh_nDelta: 45

# # Quadratic Settings:
# degree: 2
# dm_plex_box_faces: 20,30,1
# platemesh:
#   modify_mesh: true
#   nDelta: 22
#   growth: 1.1664 # 1.08^2

stab: 'supg'

dm_plex_box_lower: 0,0,0
dm_plex_box_upper: 4.2e-3,4.2e-3,5.e-5
dm_plex_dim: 3
# Faces labeled 1=z- 2=z+ 3=y- 4=y+ 5=x+ 6=x-
bc_slip_z: 1,2
bc_wall: 3
wall_comps: 1,2,3
bc_inflow: 6
bc_outflow: 5,4
gravity: 0,0,0

stg:
  use: false
  inflow_path: "./STGInflow_blasius.dat"
  mean_only: true

# ts_monitor_turbulence_spanstats:
#   collect_interval: 1
#   viewer_interval: 5
#   viewer: cgns:stats-%d.cgns
#   viewer_cgns_batch_size: 1
```

STG Inflow for Flat Plate

Using the STG Inflow for the blasius problem adds the following command-line options:

Table 4.14: Blasius Runtime Options

Option	Description	Default value	Unit
`-stg_inflow_path`	Path to the STGInflow file	`./STGInflow.dat`	
`-stg_rand_path`	Path to the STGRand file	`./STGRand.dat`	
`-stg_alpha`	Growth rate of the wavemodes	`1.01`	
`-stg_u0`	Convective velocity, U_0	`0.0`	`m/s`
`-stg_mean_only`	Only impose the mean velocity (no fluctutations)	`false`	
`-stg_strong`	Strongly enforce the STG inflow boundary condition	`false`	
`-stg_fluctuating_IC`	"Extrude" the fluctuations through the domain as an initial condition	`false`	

This problem can be run with the `blasius.yaml` file via:

```
./navierstokes -options_file blasius.yaml -stg_use true
```

Note the added `-stg_use true` flag This overrides the `stg: use: false` setting in the `blasius.yaml` file, enabling the use of the STG inflow.

4.5.2 The Navier-Stokes equations

The mathematical formulation (from [SHJ91]) is given in what follows. The compressible Navier-Stokes equations in conservative form are

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot \boldsymbol{U} &= 0 \\ \frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \left(\frac{\boldsymbol{U} \otimes \boldsymbol{U}}{\rho} + P \boldsymbol{I}_3 - \boldsymbol{\sigma} \right) - \rho \boldsymbol{b} &= 0 \\ \frac{\partial E}{\partial t} + \nabla \cdot \left(\frac{(E + P)\boldsymbol{U}}{\rho} - \boldsymbol{u} \cdot \boldsymbol{\sigma} - k \nabla T \right) - \rho \boldsymbol{b} \cdot \boldsymbol{u} &= 0 \, , \end{aligned} \tag{4.14}$$

where $\boldsymbol{\sigma} = \mu(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T + \lambda(\nabla \cdot \boldsymbol{u})\boldsymbol{I}_3)$ is the Cauchy (symmetric) stress tensor, with μ the dynamic viscosity coefficient, and $\lambda = -2/3$ the Stokes hypothesis constant. In equations (4.14), ρ represents the volume mass density, U the momentum density (defined as $\boldsymbol{U} = \rho \boldsymbol{u}$, where $\boldsymbol{u}$ is the vector velocity field), E the total energy density (defined as $E = \rho e$, where e is the total energy including thermal and kinetic but not potential energy), $\boldsymbol{I}_3$ represents the 3×3 identity matrix, $\boldsymbol{b}$ is a body force vector (e.g., gravity vector $\boldsymbol{g}$), k the thermal conductivity constant, T represents the temperature, and P the pressure, given by the following equation of state

$$P = \left(c_p / c_v - 1 \right) \left(E - \boldsymbol{U} \cdot \boldsymbol{U} / (2 \rho) \right) \, , \tag{4.15}$$

where c_p is the specific heat at constant pressure and c_v is the specific heat at constant volume (that define $\gamma = c_p / c_v$, the specific heat ratio).

The system (4.14) can be rewritten in vector form

$$\frac{\partial \boldsymbol{q}}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{q}) - S(\boldsymbol{q}) = 0 \, , \tag{4.16}$$

for the state variables 5-dimensional vector

$$\boldsymbol{q} = \begin{pmatrix} \rho \\ \boldsymbol{U} \equiv \rho \boldsymbol{u} \\ E \equiv \rho e \end{pmatrix} \begin{array}{l} \leftarrow \text{ volume mass density} \\ \leftarrow \text{ momentum density} \\ \leftarrow \text{ energy density} \end{array}$$

where the flux and the source terms, respectively, are given by

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{q}) &= \underbrace{\begin{pmatrix} \boldsymbol{U} \\ (\boldsymbol{U} \otimes \boldsymbol{U})/\rho + P\boldsymbol{I}_3 \\ (E + P)\boldsymbol{U}/\rho \end{pmatrix}}_{\boldsymbol{F}_{\text{adv}}} + \underbrace{\begin{pmatrix} 0 \\ -\boldsymbol{\sigma} \\ -\boldsymbol{u} \cdot \boldsymbol{\sigma} - k\nabla T \end{pmatrix}}_{\boldsymbol{F}_{\text{diff}}}, \\ S(\boldsymbol{q}) &= \begin{pmatrix} 0 \\ \rho \boldsymbol{b} \\ \rho \boldsymbol{b} \cdot \boldsymbol{u} \end{pmatrix}. \end{aligned} \tag{4.17}$$

4.5.2.1 Finite Element Formulation (Spatial Discretization)

Let the discrete solution be

$$\boldsymbol{q}_N(\boldsymbol{x}, t)^{(e)} = \sum_{k=1}^{P} \psi_k(\boldsymbol{x}) \boldsymbol{q}_k^{(e)}$$

with $P = p+1$ the number of nodes in the element e. We use tensor-product bases $\psi_{kji} = h_i(X_0)h_j(X_1)h_k(X_2)$.

To obtain a finite element discretization, we first multiply the strong form (4.16) by a test function $\boldsymbol{v} \in H^1(\Omega)$ and integrate,

$$\int_\Omega \boldsymbol{v} \cdot \left(\frac{\partial \boldsymbol{q}_N}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{q}_N) - \boldsymbol{S}(\boldsymbol{q}_N) \right) dV = 0 \,, \; \forall \boldsymbol{v} \in \mathcal{V}_p \,,$$

with $\mathcal{V}_p = \{ \boldsymbol{v}(\boldsymbol{x}) \in H^1(\Omega_e) \,|\, \boldsymbol{v}(\boldsymbol{x}_e(\boldsymbol{X})) \in P_p(\boldsymbol{I}), e = 1, \ldots, N_e \}$ a mapped space of polynomials containing at least polynomials of degree p (with or without the higher mixed terms that appear in tensor product spaces).

Integrating by parts on the divergence term, we arrive at the weak form,

$$\begin{aligned} \int_\Omega \boldsymbol{v} \cdot \left(\frac{\partial \boldsymbol{q}_N}{\partial t} - \boldsymbol{S}(\boldsymbol{q}_N) \right) dV - \int_\Omega \nabla \boldsymbol{v} \!:\! \boldsymbol{F}(\boldsymbol{q}_N) \, dV & \\ + \int_{\partial \Omega} \boldsymbol{v} \cdot \boldsymbol{F}(\boldsymbol{q}_N) \cdot \hat{\boldsymbol{n}} \, dS = 0 \,, \; \forall \boldsymbol{v} \in \mathcal{V}_p \,, & \end{aligned} \tag{4.18}$$

where $\boldsymbol{F}(\boldsymbol{q}_N) \cdot \hat{\boldsymbol{n}}$ is typically replaced with a boundary condition.

Note: The notation $\nabla \boldsymbol{v} \!:\! \boldsymbol{F}$ represents contraction over both fields and spatial dimensions while a single dot represents contraction in just one, which should be clear from context, e.g., $\boldsymbol{v} \cdot \boldsymbol{S}$ contracts over fields while $\boldsymbol{F} \cdot \hat{\boldsymbol{n}}$ contracts over spatial dimensions.

4.5.2.2 Time Discretization

For the time discretization, we use two types of time stepping schemes through PETSc.

Explicit time-stepping method

The following explicit formulation is solved with the adaptive Runge-Kutta-Fehlberg (RKF4-5) method by default (any explicit time-stepping scheme available in PETSc can be chosen at runtime)

$$\boldsymbol{q}_N^{n+1} = \boldsymbol{q}_N^n + \Delta t \sum_{i=1}^{s} b_i k_i \,,$$

where

$$\begin{aligned}
k_1 &= f(t^n, \boldsymbol{q}_N^n) \\
k_2 &= f(t^n + c_2 \Delta t, \boldsymbol{q}_N^n + \Delta t (a_{21} k_1)) \\
k_3 &= f(t^n + c_3 \Delta t, \boldsymbol{q}_N^n + \Delta t (a_{31} k_1 + a_{32} k_2)) \\
&\vdots \\
k_i &= f\left(t^n + c_i \Delta t, \boldsymbol{q}_N^n + \Delta t \sum_{j=1}^{s} a_{ij} k_j \right)
\end{aligned}$$

and with

$$f(t^n, \boldsymbol{q}_N^n) = -[\nabla \cdot \boldsymbol{F}(\boldsymbol{q}_N)]^n + [S(\boldsymbol{q}_N)]^n \,.$$

Implicit time-stepping method

This time stepping method which can be selected using the option `-implicit` is solved with Backward Differentiation Formula (BDF) method by default (similarly, any implicit time-stepping scheme available in PETSc can be chosen at runtime). The implicit formulation solves nonlinear systems for $\boldsymbol{q}_N$:

$$\boldsymbol{f}(\boldsymbol{q}_N) \equiv \boldsymbol{g}(t^{n+1}, \boldsymbol{q}_N, \dot{\boldsymbol{q}}_N) = 0 \,, \tag{4.19}$$

where the time derivative $\dot{\boldsymbol{q}}_N$ is defined by

$$\dot{\boldsymbol{q}}_N(\boldsymbol{q}_N) = \alpha \boldsymbol{q}_N + \boldsymbol{z}_N$$

in terms of $\boldsymbol{z}_N$ from prior state and $\alpha > 0$, both of which depend on the specific time integration scheme (backward difference formulas, generalized alpha, implicit Runge-Kutta, etc.). Each nonlinear system (4.19) will correspond to a weak form, as explained below. In determining how difficult a given problem is to solve, we consider the Jacobian of (4.19),

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{q}_N} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{q}_N} + \alpha \frac{\partial \boldsymbol{g}}{\partial \dot{\boldsymbol{q}}_N}.$$

The scalar "shift" α scales inversely with the time step Δt, so small time steps result in the Jacobian being dominated by the second term, which is a sort of "mass matrix", and typically well-conditioned independent of grid resolution with a simple preconditioner (such as Jacobi). In contrast, the first term dominates for large time steps, with a condition number that grows with the diameter of the domain and polynomial degree of the approximation space. Both terms are significant for time-accurate simulation and the setup costs of strong preconditioners must be balanced with the convergence rate of Krylov methods using weak preconditioners.

More details of PETSc's time stepping solvers can be found in the TS User Guide.

4.5.2.3 Stabilization

We solve (4.18) using a Galerkin discretization (default) or a stabilized method, as is necessary for most real-world flows.

Galerkin methods produce oscillations for transport-dominated problems (any time the cell Péclet number is larger than 1), and those tend to blow up for nonlinear problems such as the Euler equations and (low-viscosity/poorly resolved) Navier-Stokes, in which case stabilization is necessary. Our formulation follows [HST10], which offers a comprehensive review of stabilization and shock-capturing methods for continuous finite element discretization of compressible flows.

- **SUPG** (streamline-upwind/Petrov-Galerkin)

  In this method, the weighted residual of the strong form (4.16) is added to the Galerkin formulation (4.18). The weak form for this method is given as

$$\begin{aligned} \int_\Omega \boldsymbol{v} \cdot \left(\frac{\partial \boldsymbol{q}_N}{\partial t} - \boldsymbol{S}(\boldsymbol{q}_N) \right) dV - \int_\Omega \nabla \boldsymbol{v} \!:\! \boldsymbol{F}(\boldsymbol{q}_N)\, dV & \\ + \int_{\partial \Omega} \boldsymbol{v} \cdot \boldsymbol{F}(\boldsymbol{q}_N) \cdot \hat{\boldsymbol{n}}\, dS & \\ + \int_\Omega \nabla \boldsymbol{v} \!:\! \left(\frac{\partial \boldsymbol{F}_{\text{adv}}}{\partial \boldsymbol{q}} \right) \boldsymbol{\tau} \left(\frac{\partial \boldsymbol{q}_N}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{q}_N) - \boldsymbol{S}(\boldsymbol{q}_N) \right) dV = 0, \ \forall \boldsymbol{v} \in \mathcal{V}_p & \end{aligned} \tag{4.20}$$

  This stabilization technique can be selected using the option `-stab supg`.

- **SU** (streamline-upwind)

  This method is a simplified version of *SUPG* (4.20) which is developed for debugging/comparison purposes. The weak form for this method is

$$\begin{aligned} \int_\Omega \boldsymbol{v} \cdot \left(\frac{\partial \boldsymbol{q}_N}{\partial t} - \boldsymbol{S}(\boldsymbol{q}_N) \right) dV - \int_\Omega \nabla \boldsymbol{v} \!:\! \boldsymbol{F}(\boldsymbol{q}_N)\, dV & \\ + \int_{\partial \Omega} \boldsymbol{v} \cdot \boldsymbol{F}(\boldsymbol{q}_N) \cdot \hat{\boldsymbol{n}}\, dS & \\ + \int_\Omega \nabla \boldsymbol{v} \!:\! \left(\frac{\partial \boldsymbol{F}_{\text{adv}}}{\partial \boldsymbol{q}} \right) \boldsymbol{\tau} \nabla \cdot \boldsymbol{F}(\boldsymbol{q}_N)\, dV = 0, \ \forall \boldsymbol{v} \in \mathcal{V}_p & \end{aligned} \tag{4.21}$$

  This stabilization technique can be selected using the option `-stab su`.

In both (4.21) and (4.20), $\boldsymbol{\tau} \in \mathbb{R}^{5\times 5}$ (field indices) is an intrinsic time scale matrix. The SUPG technique and the operator $\frac{\partial \boldsymbol{F}_{\text{adv}}}{\partial \boldsymbol{q}}$ (rather than its transpose) can be explained via an ansatz for subgrid state fluctuations $\tilde{\boldsymbol{q}} = -\boldsymbol{\tau}\boldsymbol{r}$ where $\boldsymbol{r}$ is a strong form residual. The forward variational form can be readily expressed by differentiating $\boldsymbol{F}_{\text{adv}}$ of (4.17)

$$\begin{aligned} \mathrm{d}\boldsymbol{F}_{\text{adv}}(\mathrm{d}\boldsymbol{q}; \boldsymbol{q}) &= \frac{\partial \boldsymbol{F}_{\text{adv}}}{\partial \boldsymbol{q}} \mathrm{d}\boldsymbol{q} \\ &= \begin{pmatrix} \mathrm{d}\boldsymbol{U} \\ (\mathrm{d}\boldsymbol{U} \otimes \boldsymbol{U} + \boldsymbol{U} \otimes \mathrm{d}\boldsymbol{U})/\rho - (\boldsymbol{U} \otimes \boldsymbol{U})/\rho^2\, \mathrm{d}\rho + \mathrm{d}P \boldsymbol{I}_3 \\ (E + P)\, \mathrm{d}\boldsymbol{U}/\rho + (\mathrm{d}E + \mathrm{d}P)\boldsymbol{U}/\rho - (E + P)\boldsymbol{U}/\rho^2\, \mathrm{d}\rho \end{pmatrix}, \end{aligned}$$

where $\mathrm{d}P$ is defined by differentiating (4.15).

Stabilization scale τ

A velocity vector $\boldsymbol{u}$ can be pulled back to the reference element as $\boldsymbol{u}_{\boldsymbol{X}} = \nabla_x \boldsymbol{X} \cdot \boldsymbol{u}$, with units of reference length (non-dimensional) per second. To build intuition, consider a boundary layer element of dimension $(1, \epsilon)$, for which $\nabla_x \boldsymbol{X} = \begin{pmatrix} 2 & \\ & 2/\epsilon \end{pmatrix}$. So a small normal component of velocity will be amplified (by a factor of the aspect ratio $1/\epsilon$) in this transformation. The ratio $\|\boldsymbol{u}\| / \|\boldsymbol{u}_{\boldsymbol{X}}\|$ is a covariant measure of (half) the element length in the direction of the velocity. A contravariant measure of element length in the direction of a unit vector $\hat{\boldsymbol{n}}$ is given by $\|(\nabla_{\boldsymbol{X}} \boldsymbol{x})^T \hat{\boldsymbol{n}}\|$. While $\nabla_{\boldsymbol{X}} \boldsymbol{x}$ is readily computable, its inverse $\nabla_x \boldsymbol{X}$ is needed directly in finite element methods and thus more convenient for our use. If we consider a parallelogram, the covariant measure is larger than the contravariant measure for vectors pointing between acute corners and the opposite holds for vectors between oblique corners.

The cell Péclet number is classically defined by $\mathrm{Pe}_h = \|\boldsymbol{u}\| h / (2\kappa)$ where κ is the diffusivity (units of m^2/s). This can be generalized to arbitrary grids by defining the local Péclet number

$$\mathrm{Pe} = \frac{\|\boldsymbol{u}\|^2}{\|\boldsymbol{u}_{\boldsymbol{X}}\| \kappa}. \tag{4.22}$$

For scalar advection-diffusion, the stabilization is a scalar

$$\tau = \frac{\xi(\mathrm{Pe})}{\|\boldsymbol{u}_{\boldsymbol{X}}\|}, \tag{4.23}$$

where $\xi(\mathrm{Pe}) = \coth \mathrm{Pe} - 1/\mathrm{Pe}$ approaches 1 at large local Péclet number. Note that τ has units of time and, in the transport-dominated limit, is proportional to element transit time in the direction of the propagating wave. For advection-diffusion, $\boldsymbol{F}(q) = \boldsymbol{u} q$, and thus the SU stabilization term is

$$\nabla v \cdot \boldsymbol{u} \tau \boldsymbol{u} \cdot \nabla q = \nabla_{\boldsymbol{X}} v \cdot (\boldsymbol{u}_{\boldsymbol{X}} \tau \boldsymbol{u}_{\boldsymbol{X}}) \cdot \nabla_{\boldsymbol{X}} q. \tag{4.24}$$

where the term in parentheses is a rank-1 diffusivity tensor that has been pulled back to the reference element. See [HST10] equations 15-17 and 34-36 for further discussion of this formulation.

For the Navier-Stokes and Euler equations, [WJD03] defines a 5×5 diagonal stabilization $\mathrm{diag}(\tau_c, \tau_m, \tau_m, \tau_m, \tau_E)$ consisting of

1. continuity stabilization τ_c
2. momentum stabilization τ_m
3. energy stabilization τ_E

The Navier-Stokes code in this example uses the following formulation for τ_c, τ_m, τ_E:

$$\begin{aligned} \tau_c &= \frac{C_c \mathcal{F}}{8\rho \, \mathrm{trace}(\boldsymbol{g})} \\ \tau_m &= \frac{C_m}{\mathcal{F}} \\ \tau_E &= \frac{C_E}{\mathcal{F} c_v} \end{aligned}$$

$$\mathcal{F} = \sqrt{\rho^2 \left[\left(\frac{2C_t}{\Delta t} \right)^2 + \boldsymbol{u} \cdot (\boldsymbol{u} \cdot \boldsymbol{g}) + C_v \mu^2 \|\boldsymbol{g}\|_F^2 \right]}$$

where $\boldsymbol{g} = \nabla_x \boldsymbol{X} \cdot \nabla_x \boldsymbol{X}$ is the metric tensor and $\| \cdot \|_F$ is the Frobenius norm. This formulation is currently not available in the Euler code.

In the Euler code, we follow [HST10] in defining a 3×3 diagonal stabilization according to spatial criterion 2 (equation 27) as follows.

$$\tau_{ii} = c_\tau \frac{2\xi(\mathrm{Pe})}{(\lambda_{\max\mathrm{abs}})_i \|\nabla_{x_i} \boldsymbol{X}\|} \tag{4.25}$$

where c_τ is a multiplicative constant reported to be optimal at 0.5 for linear elements, $\hat{\boldsymbol{n}}_i$ is a unit vector in direction i, and $\nabla_{x_i} = \hat{\boldsymbol{n}}_i \cdot \nabla_{\boldsymbol{x}}$ is the derivative in direction i. The flux Jacobian $\frac{\partial \boldsymbol{F}_{\mathrm{adv}}}{\partial \boldsymbol{q}} \cdot \hat{\boldsymbol{n}}_i$ in each direction i is a 5×5 matrix with spectral radius $(\lambda_{\max\mathrm{abs}})_i$ equal to the fastest wave speed. The complete set of eigenvalues of the Euler flux Jacobian in direction i are (e.g., [Tor09])

$$\Lambda_i = [u_i - a, u_i, u_i, u_i, u_i + a], \tag{4.26}$$

where $u_i = \boldsymbol{u} \cdot \hat{\boldsymbol{n}}_i$ is the velocity component in direction i and $a = \sqrt{\gamma P/\rho}$ is the sound speed for ideal gasses. Note that the first and last eigenvalues represent nonlinear acoustic waves while the middle three are linearly degenerate, carrying a contact wave (temperature) and transverse components of momentum. The fastest wave speed in direction i is thus

$$\lambda_{\max\mathrm{abs}}\Big(\frac{\partial \boldsymbol{F}_{\mathrm{adv}}}{\partial \boldsymbol{q}} \cdot \hat{\boldsymbol{n}}_i\Big) = |u_i| + a \tag{4.27}$$

Note that this wave speed is specific to ideal gases as γ is an ideal gas parameter; other equations of state will yield a different acoustic wave speed.

Currently, this demo provides three types of problems/physical models that can be selected at run time via the option `-problem`. *Differential Filtering*, the problem of the transport of energy in a uniform vector velocity field, *Isentropic Vortex*, the exact solution to the Euler equations, and the so called *Gaussian Wave* problem.

4.5.2.4 Subgrid Stress Modeling

When a fluid simulation is under-resolved (the smallest length scale resolved by the grid is much larger than the smallest physical scale, the Kolmogorov length scale), this is mathematically interpreted as filtering the Navier-Stokes equations. This is known as large-eddy simulation (LES), as only the "large" scales of turbulence are resolved. This filtering operation results in an extra stress-like term, $\boldsymbol{\tau}^r$, representing the effect of unresolved (or "subgrid" scale) structures in the flow. Denoting the filtering operation by $\bar{\cdot}$, the LES governing equations are:

$$\frac{\partial \bar{\boldsymbol{q}}}{\partial t} + \nabla \cdot \bar{\boldsymbol{F}}(\bar{\boldsymbol{q}}) - S(\bar{\boldsymbol{q}}) = 0\,, \tag{4.28}$$

where

$$\bar{\boldsymbol{F}}(\bar{\boldsymbol{q}}) = \boldsymbol{F}(\bar{\boldsymbol{q}}) + \begin{pmatrix} 0 \\ \boldsymbol{\tau}^r \\ \boldsymbol{u} \cdot \boldsymbol{\tau}^r \end{pmatrix} \tag{4.29}$$

More details on deriving the above expression, filtering, and large eddy simulation can be found in [Pop00]. To close the problem, the subgrid stress must be defined. For implicit LES, the subgrid stress is set to zero and the numerical properties of the discretized system are assumed to account for the effect of subgrid scale structures on the filtered solution field. For explicit LES, it is defined by a subgrid stress model.

Data-driven SGS Model

The data-driven SGS model implemented here uses a small neural network to compute the SGS term. The SGS tensor is calculated at nodes using an L^2 projection of the velocity gradient and grid anisotropy tensor, and then interpolated onto quadrature points. More details regarding the theoretical background of the model can be found in [PJE22a] and [PJE22b].

The neural network itself consists of 1 hidden layer and 20 neurons, using Leaky ReLU as its activation function. The slope parameter for the Leaky ReLU function is set via `-sgs_model_dd_leakyrelu_alpha`. The outputs of the network are assumed to be normalized on a min-max scale, so they must be rescaled by the original min-max bounds. Parameters for the neural network are put into files in a directory found in `-sgs_model_dd_parameter_dir`. These files store the network weights (`w1.dat` and `w2.dat`), biases (`b1.dat` and `b2.dat`), and scaling parameters (`OutScaling.dat`). The first row of each files stores the number of columns and rows in each file. Note that the weight coefficients are assumed to be in column-major order. This is done to keep consistent with legacy file compatibility.

Note: The current data-driven model parameters are not accurate and are for regression testing only.

4.5.2.5 Differential Filtering

There is the option to filter the solution field using differential filtering. This was first proposed in [Ger86], using an inverse Hemholtz operator. The strong form of the differential equation is

$$\overline{\phi} - \nabla \cdot (\beta (\boldsymbol{D}\boldsymbol{\Delta})^2 \nabla \overline{\phi}) = \phi$$

for ϕ the scalar solution field we want to filter, $\overline{\phi}$ the filtered scalar solution field, $\boldsymbol{\Delta} \in \mathbb{R}^{3\times3}$ a symmetric positive-definite rank 2 tensor defining the width of the filter, $\boldsymbol{D}$ is the filter width scaling tensor (also a rank 2 SPD tensor), and β is a kernel scaling factor on the filter tensor. This admits the weak form:

$$\int_\Omega \left(v\overline{\phi} + \beta \nabla v \cdot (\boldsymbol{D}\boldsymbol{\Delta})^2 \nabla \overline{\phi} \right) d\Omega - \cancel{\int_{\partial\Omega} \beta v \nabla \overline{\phi} \cdot (\boldsymbol{D}\boldsymbol{\Delta})^2 \hat{\boldsymbol{n}} \, d\partial\Omega} = \int_\Omega v\phi \, , \; \forall v \in \mathcal{V}_p$$

The boundary integral resulting from integration-by-parts is crossed out, as we assume that $(\boldsymbol{D}\boldsymbol{\Delta})^2 = \boldsymbol{0} \Leftrightarrow \overline{\phi} = \phi$ at boundaries (this is reasonable at walls, but for convenience elsewhere).

Filter width tensor, Δ

For homogenous filtering, $\boldsymbol{\Delta}$ is defined as the identity matrix.

Note: It is common to denote a filter width dimensioned relative to the radial distance of the filter kernel. Note here we use the filter *diameter* instead, as that feels more natural (albeit mathematically less convenient). For example, under this definition a box filter would be defined as:

$$B(\Delta; \boldsymbol{r}) = \begin{cases} 1 & \|\boldsymbol{r}\| \leq \Delta/2 \\ 0 & \|\boldsymbol{r}\| > \Delta/2 \end{cases}$$

For inhomogeneous anisotropic filtering, we use the finite element grid itself to define $\boldsymbol{\Delta}$. This is set via `-diff_filter_grid_based_width`. Specifically, we use the filter width tensor defined in [PJE22b]. For finite element grids, the filter width tensor is most conveniently defined by $\boldsymbol{\Delta} = \boldsymbol{g}^{-1/2}$ where $\boldsymbol{g} = \nabla_x \boldsymbol{X} \cdot \nabla_x \boldsymbol{X}$ is the metric tensor.

Filter width scaling tensor, D

The filter width tensor $\boldsymbol{\Delta}$, be it defined from grid based sources or just the homogenous filtering, can be scaled anisotropically. The coefficients for that anisotropic scaling are given by `-diff_filter_width_scaling`, denoted here by c_1, c_2, c_3. The definition for $\boldsymbol{D}$ then becomes

$$\boldsymbol{D} = \begin{bmatrix} c_1 & 0 & 0 \\ 0 & c_2 & 0 \\ 0 & 0 & c_3 \end{bmatrix}$$

In the case of $\boldsymbol{\Delta}$ being defined as homogenous, $\boldsymbol{D\Delta}$ means that $\boldsymbol{D}$ effectively sets the filter width.

The filtering at the wall may also be damped, to smoothly meet the $\overline{\phi} = \phi$ boundary condition at the wall. The selected damping function for this is the van Driest function [VD56]:

$$\zeta = 1 - \exp\left(-\frac{y^+}{A^+}\right)$$

where y^+ is the wall-friction scaled wall-distance ($y^+ = yu_\tau/\nu = y/\delta_\nu$), A^+ is some wall-friction scaled scale factor, and ζ is the damping coefficient. For this implementation, we assume that δ_ν is constant across the wall and is defined by `-diff_filter_friction_length`. A^+ is defined by `-diff_filter_damping_constant`.

To apply this scalar damping coefficient to the filter width tensor, we construct the wall-damping tensor from it. The construction implemented currently limits damping in the wall parallel directions to be no less than the original filter width defined by $\boldsymbol{\Delta}$. The wall-normal filter width is allowed to be damped to a zero filter width. It is currently assumed that the second component of the filter width tensor is in the wall-normal direction. Under these assumptions, $\boldsymbol{D}$ then becomes:

$$\boldsymbol{D} = \begin{bmatrix} \max(1, \zeta c_1) & 0 & 0 \\ 0 & \zeta c_2 & 0 \\ 0 & 0 & \max(1, \zeta c_3) \end{bmatrix}$$

Filter kernel scaling, β

While we define $\boldsymbol{D\Delta}$ to be of a certain physical filter width, the actual width of the implied filter kernel is quite larger than "normal" kernels. To account for this, we use β to scale the filter tensor to the appropriate size, as is done in [BJ16]. To match the "size" of a normal kernel to our differential kernel, we attempt to have them match second order moments with respect to the prescribed filter width. To match the box and Gaussian filters "sizes", we use $\beta = 1/10$ and $\beta = 1/6$, respectively. β can be set via `-diff_filter_kernel_scaling`.

4.5.3 Advection

A simplified version of system (4.14), only accounting for the transport of total energy, is given by

$$\frac{\partial E}{\partial t} + \nabla \cdot (\boldsymbol{u}E) = 0\,, \tag{4.30}$$

with $\boldsymbol{u}$ the vector velocity field. In this particular test case, a blob of total energy (defined by a characteristic radius r_c) is transported by two different wind types.

- **Rotation**

  In this case, a uniform circular velocity field transports the blob of total energy. We have solved (4.30) applying zero energy density E, and no-flux for $\boldsymbol{u}$ on the boundaries.

- **Translation**

  In this case, a background wind with a constant rectilinear velocity field, enters the domain and transports the blob of total energy out of the domain.

  For the inflow boundary conditions, a prescribed E_{wind} is applied weakly on the inflow boundaries such that the weak form boundary integral in (4.18) is defined as

$$\int_{\partial\Omega_{inflow}} \boldsymbol{v} \cdot \boldsymbol{F}(\boldsymbol{q}_N) \cdot \hat{\boldsymbol{n}} \, dS = \int_{\partial\Omega_{inflow}} \boldsymbol{v} \, E_{wind} \, \boldsymbol{u} \cdot \hat{\boldsymbol{n}} \, dS \, ,$$

  For the outflow boundary conditions, we have used the current values of E, following [PMK92] which extends the validity of the weak form of the governing equations to the outflow instead of replacing them with unknown essential or natural boundary conditions. The weak form boundary integral in (4.18) for outflow boundary conditions is defined as

$$\int_{\partial\Omega_{outflow}} \boldsymbol{v} \cdot \boldsymbol{F}(\boldsymbol{q}_N) \cdot \hat{\boldsymbol{n}} \, dS = \int_{\partial\Omega_{outflow}} \boldsymbol{v} \, E \, \boldsymbol{u} \cdot \hat{\boldsymbol{n}} \, dS \, ,$$

4.5.4 Isentropic Vortex

Three-dimensional Euler equations, which are simplified and nondimensionalized version of system (4.14) and account only for the convective fluxes, are given by

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot \boldsymbol{U} &= 0 \\ \frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \left(\frac{\boldsymbol{U} \otimes \boldsymbol{U}}{\rho} + P \boldsymbol{I}_3 \right) &= 0 \\ \frac{\partial E}{\partial t} + \nabla \cdot \left(\frac{(E + P)\boldsymbol{U}}{\rho} \right) &= 0 \, , \end{aligned} \tag{4.31}$$

Following the setup given in [ZZS11], the mean flow for this problem is $\rho = 1$, $P = 1$, $T = P/\rho = 1$ (Specific Gas Constant, R, is 1), and $\boldsymbol{u} = (u_1, u_2, 0)$ while the perturbation $\delta \boldsymbol{u}$, and δT are defined as

$$\begin{aligned} (\delta u_1, \, \delta u_2) &= \frac{\epsilon}{2\pi} \, e^{0.5(1-r^2)} \, (-\bar{y}, \, \bar{x}) \, , \\ \delta T &= - \frac{(\gamma - 1) \, \epsilon^2}{8 \, \gamma \, \pi^2} \, e^{1-r^2} \, , \end{aligned}$$

where $(\bar{x}, \, \bar{y}) = (x - x_c, \, y - y_c)$, $(x_c, \, y_c)$ represents the center of the domain, $r^2 = \bar{x}^2 + \bar{y}^2$, and ϵ is the vortex strength ($\epsilon < 10$). There is no perturbation in the entropy $S = P/\rho^\gamma$ ($\delta S = 0$).

4.5.5 Shock Tube

This test problem is based on Sod's Shock Tube (from[sod]), a canonical test case for discontinuity capturing in one dimension. For this problem, the three-dimensional Euler equations are formulated exactly as in the Isentropic Vortex problem. The default initial conditions are $P = 1$, $\rho = 1$ for the driver section and $P = 0.1$, $\rho = 0.125$ for the driven section. The initial velocity is zero in both sections. Slip boundary conditions are applied to the side walls and wall boundary conditions are applied at the end walls.

SU upwinding and discontinuity capturing have been implemented into the explicit timestepping operator for this problem. Discontinuity capturing is accomplished using a modified version of the $YZ\beta$ operator described in [TS07]. This discontinuity capturing scheme involves the introduction of a dissipation term of the form

$$\int_{\Omega} \nu_{SHOCK} \nabla \boldsymbol{v} \colon \nabla \boldsymbol{q} dV$$

The shock capturing viscosity is implemented following the first formulation described in [TS07]. The characteristic velocity u_{cha} is taken to be the acoustic speed while the reference density ρ_{ref} is just the local density. Shock capturing viscosity is defined by the following

$$\nu_{SHOCK} = \tau_{SHOCK} u_{cha}^2$$

where,

$$\tau_{SHOCK} = \frac{h_{SHOCK}}{2u_{cha}} \left(\frac{|\nabla \rho| h_{SHOCK}}{\rho_{ref}} \right)^{\beta}$$

β is a tuning parameter set between 1 (smoother shocks) and 2 (sharper shocks. The parameter h_{SHOCK} is a length scale that is proportional to the element length in the direction of the density gradient unit vector. This density gradient unit vector is defined as $\hat{\boldsymbol{j}} = \frac{\nabla \rho}{|\nabla \rho|}$. The original formulation of Tezduyar and Senga relies on the shape function gradient to define the element length scale, but this gradient is not available to qFunctions in libCEED. To avoid this problem, h_{SHOCK} is defined in the current implementation as

$$h_{SHOCK} = 2 \left(C_{YZB} \, |\boldsymbol{p}| \right)^{-1}$$

where

$$p_k = \hat{j}_i \frac{\partial \xi_i}{x_k}$$

The constant C_{YZB} is set to 0.1 for piecewise linear elements in the current implementation. Larger values approaching unity are expected with more robust stabilization and implicit timestepping.

4.5.6 Gaussian Wave

This test case is taken/inspired by that presented in [MDGP+14]. It is intended to test non-reflecting/Riemann boundary conditions. It's primarily intended for Euler equations, but has been implemented for the Navier-Stokes equations here for flexibility.

The problem has a perturbed initial condition and lets it evolve in time. The initial condition contains a Gaussian perturbation in the pressure field:

$$\begin{aligned}
\rho &= \rho_\infty \left(1 + A \exp \left(\frac{-(\bar{x}^2 + \bar{y}^2)}{2\sigma^2} \right) \right) \\
\boldsymbol{U} &= \boldsymbol{U}_\infty \\
E &= \frac{p_\infty}{\gamma - 1} \left(1 + A \exp \left(\frac{-(\bar{x}^2 + \bar{y}^2)}{2\sigma^2} \right) \right) + \frac{\boldsymbol{U}_\infty \cdot \boldsymbol{U}_\infty}{2\rho_\infty},
\end{aligned}$$

where A and σ are the amplitude and width of the perturbation, respectively, and $(\bar{x}, \bar{y}) = (x - x_e, y - y_e)$ is the distance to the epicenter of the perturbation, (x_e, y_e). The simulation produces a strong acoustic wave and leaves behind a cold thermal bubble that advects at the fluid velocity.

The boundary conditions are freestream in the x and y directions. When using an HLL (Harten, Lax, van Leer) Riemann solver [Tor09] (option `-freestream_riemann hll`), the acoustic waves exit the domain cleanly, but when the thermal bubble reaches the boundary, it produces strong thermal oscillations that become acoustic waves reflecting into the domain. This problem can be fixed using a more sophisticated Riemann solver such as HLLC [Tor09] (option `-freestream_riemann hllc`, which is default), which is a linear constant-pressure wave that transports temperature and transverse momentum at the fluid velocity.

4.5.7 Vortex Shedding - Flow past Cylinder

This test case, based on [SHJ91], is an example of using an externally provided mesh from Gmsh. A cylinder with diameter $D = 1$ is centered at $(0, 0)$ in a computational domain $-4.5 \leq x \leq 15.5$, $-4.5 \leq y \leq 4.5$. We solve this as a 3D problem with (default) one element in the z direction. The domain is filled with an ideal gas at rest (zero velocity) with temperature 24.92 and pressure 7143. The viscosity is 0.01 and thermal conductivity is 14.34 to maintain a Prandtl number of 0.71, which is typical for air. At time $t = 0$, this domain is subjected to freestream boundary conditions at the inflow (left) and Riemann-type outflow on the right, with exterior reference state at velocity $(1, 0, 0)$ giving Reynolds number 100 and Mach number 0.01. A symmetry (adiabatic free slip) condition is imposed at the top and bottom boundaries $(y = \pm 4.5)$ (zero normal velocity component, zero heat-flux). The cylinder wall is an adiabatic (no heat flux) no-slip boundary condition. As we evolve in time, eddies appear past the cylinder leading to a vortex shedding known as the vortex street, with shedding period of about 6.

The Gmsh input file, `examples/fluids/meshes/cylinder.geo` is parametrized to facilitate experimenting with similar configurations. The Strouhal number (nondimensional shedding frequency) is sensitive to the size of the computational domain and boundary conditions.

Forces on the cylinder walls are computed using the "reaction force" method, which is variationally consistent with the volume operator. Given the force components $\boldsymbol{F} = (F_x, F_y, F_z)$ and surface area $S = \pi D L_z$ where L_z is the spanwise extent of the domain, we define the coefficients of lift and drag as

$$C_L = \frac{2F_y}{\rho_\infty u_\infty^2 S}$$
$$C_D = \frac{2F_x}{\rho_\infty u_\infty^2 S}$$

where ρ_∞, u_∞ are the freestream (inflow) density and velocity respectively.

4.5.8 Density Current

For this test problem (from [SWW+93]), we solve the full Navier-Stokes equations (4.14), for which a cold air bubble (of radius r_c) drops by convection in a neutrally stratified atmosphere. Its initial condition is defined in terms of the Exner pressure, $\pi(\boldsymbol{x}, t)$, and potential temperature, $\theta(\boldsymbol{x}, t)$, that relate to the state variables via

$$\rho = \frac{P_0}{(c_p - c_v)\theta(\boldsymbol{x}, t)} \pi(\boldsymbol{x}, t)^{\frac{c_v}{c_p - c_v}},$$
$$e = c_v \theta(\boldsymbol{x}, t) \pi(\boldsymbol{x}, t) + \boldsymbol{u} \cdot \boldsymbol{u}/2 + gz,$$

where P_0 is the atmospheric pressure. For this problem, we have used no-slip and non-penetration boundary conditions for $\boldsymbol{u}$, and no-flux for mass and energy densities.

4.5.9 Channel

A compressible channel flow. Analytical solution given in [Whi99]:

$$u_1 = u_{\max}\left[1 - \left(\frac{x_2}{H}\right)^2\right] \qquad u_2 = u_3 = 0$$
$$T = T_w\left[1 + \frac{Pr\hat{E}c}{3}\left\{1 - \left(\frac{x_2}{H}\right)^4\right\}\right]$$
$$p = p_0 - \frac{2\rho_0 u_{\max}^2 x_1}{Re_H H}$$

where H is the channel half-height, $u_{\max}$ is the center velocity, T_w is the temperature at the wall, $Pr = \frac{\mu}{c_p \kappa}$ is the Prandlt number, $\hat{E}_c = \frac{u_{\max}^2}{c_p T_w}$ is the modified Eckert number, and $Re_h = \frac{u_{\max} H}{\nu}$ is the Reynolds number.

Boundary conditions are periodic in the streamwise direction, and no-slip and non-penetration boundary conditions at the walls. The flow is driven by a body force determined analytically from the fluid properties and setup parameters H and $u_{\max}$.

4.5.10 Flat Plate Boundary Layer

4.5.10.1 Laminar Boundary Layer - Blasius

Simulation of a laminar boundary layer flow, with the inflow being prescribed by a Blasius similarity solution. At the inflow, the velocity is prescribed by the Blasius soution profile, density is set constant, and temperature is allowed to float. Using `weakT: true`, density is allowed to float and temperature is set constant. At the outlet, a user-set pressure is used for pressure in the inviscid flux terms (all other inviscid flux terms use interior solution values). The wall is a no-slip, no-penetration, no-heat flux condition. The top of the domain is treated as an outflow and is tilted at a downward angle to ensure that flow is always exiting it.

4.5.10.2 Turbulent Boundary Layer

Simulating a turbulent boundary layer without modeling the turbulence requires resolving the turbulent flow structures. These structures may be introduced into the simulations either by allowing a laminar boundary layer naturally transition to turbulence, or imposing turbulent structures at the inflow. The latter approach has been taken here, specifically using a *synthetic turbulence generation* (STG) method.

Synthetic Turbulence Generation (STG) Boundary Condition

We use the STG method described in [SSST14]. Below follows a re-description of the formulation to match the present notation, and then a description of the implementation and usage.

Equation Formulation

$$\boldsymbol{u}(\boldsymbol{x},t) = \overline{\boldsymbol{u}}(\boldsymbol{x}) + \boldsymbol{C}(\boldsymbol{x}) \cdot \boldsymbol{v}'$$

$$\begin{aligned}\boldsymbol{v}' &= 2\sqrt{3/2}\sum_{n=1}^{N} \sqrt{q^n(\boldsymbol{x})}\boldsymbol{\sigma}^n \cos(\kappa^n \boldsymbol{d}^n \cdot \hat{\boldsymbol{x}}^n(\boldsymbol{x},t) + \phi^n) \\ \hat{\boldsymbol{x}}^n &= \left[(x - U_0 t)\max(2\kappa_{\min}/\kappa^n, 0.1), y, z\right]^T\end{aligned}$$

Here, we define the number of wavemodes N, set of random numbers $\{\boldsymbol{\sigma}^n, \boldsymbol{d}^n, \phi^n\}_{n=1}^N$, the Cholesky decomposition of the Reynolds stress tensor $\boldsymbol{C}$ (such that $\boldsymbol{R} = \boldsymbol{C}\boldsymbol{C}^T$), bulk velocity U_0, wavemode amplitude q^n, wavemode frequency κ^n, and $\kappa_{\min} = 0.5 \min_{\boldsymbol{x}}(\kappa_e)$.

$$\kappa_e = \frac{2\pi}{\min(2d_w, 3.0 l_t)}$$

where l_t is the turbulence length scale, and d_w is the distance to the nearest wall.

The set of wavemode frequencies is defined by a geometric distribution:

$$\kappa^n = \kappa_{\min}(1+\alpha)^{n-1}\,, \quad \forall n = 1,2,\dots,N$$

The wavemode amplitudes q^n are defined by a model energy spectrum $E(\kappa)$:

$$q^n = \frac{E(\kappa^n)\Delta\kappa^n}{\sum_{n=1}^{N} E(\kappa^n)\Delta\kappa^n}\,, \quad \Delta\kappa^n = \kappa^n - \kappa^{n-1}$$

$$E(\kappa) = \frac{(\kappa/\kappa_e)^4}{[1+2.4(\kappa/\kappa_e)^2]^{17/6}} f_\eta f_{\mathrm{cut}}$$

$$f_\eta = \exp\left[-(12\kappa/\kappa_\eta)^2\right], \quad f_{\mathrm{cut}} = \exp\left(-\left[\frac{4\max(\kappa - 0.9\kappa_{\mathrm{cut}}, 0)}{\kappa_{\mathrm{cut}}}\right]^3\right)$$

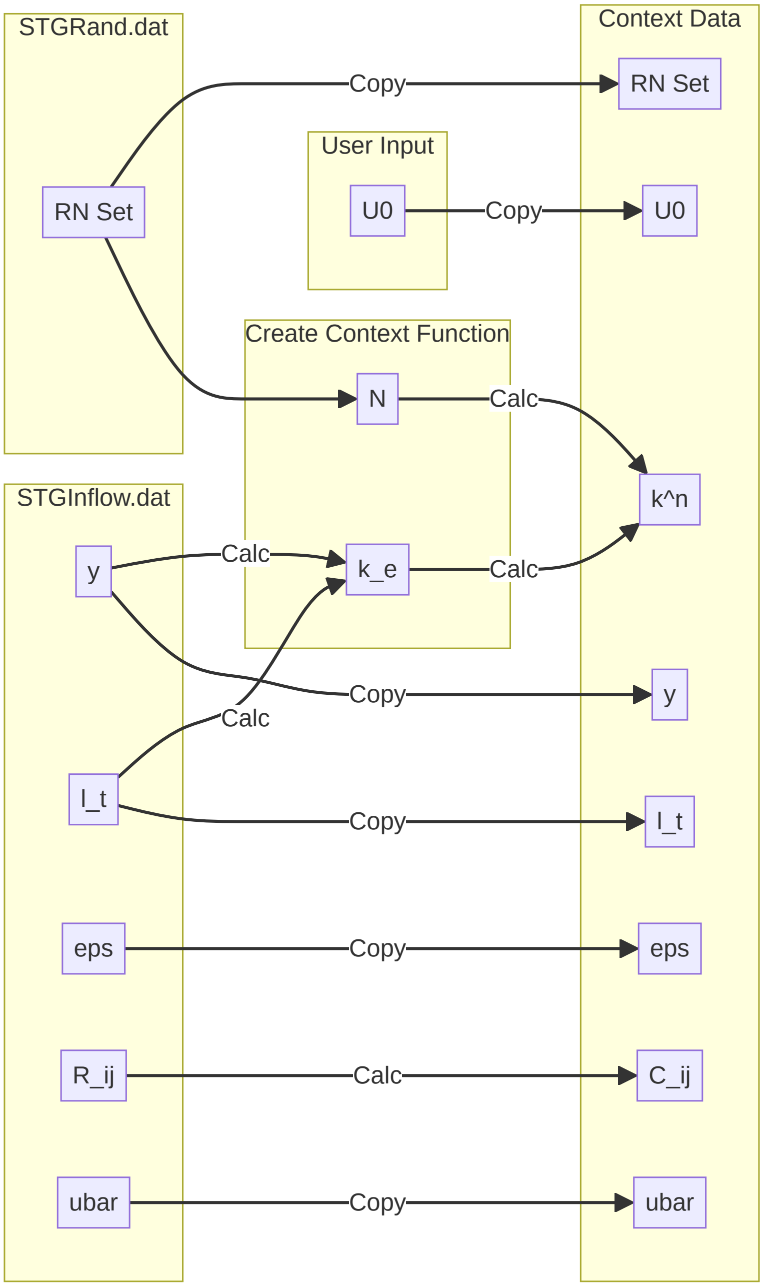
κ_η represents turbulent dissipation frequency, and is given as $2\pi(\nu^3/\varepsilon)^{-1/4}$ with ν the kinematic viscosity and ε the turbulent dissipation. κ_{cut} approximates the effective cutoff frequency of the mesh (viewing the mesh as a filter on solution over Ω) and is given by:

$$\kappa_{\mathrm{cut}} = \frac{2\pi}{2\min\{[\max(h_y, h_z, 0.3h_{\max}) + 0.1d_w], h_{\max}\}}$$

The enforcement of the boundary condition is identical to the blasius inflow; it weakly enforces velocity, with the option of weakly enforcing either density or temperature using the the `-weakT` flag.

Initialization Data Flow

Data flow for initializing function (which creates the context data struct) is given below:



This is done once at runtime. The spatially-varying terms are then evaluated at each quadrature point on-the-fly, either by interpolation (for l_t, ε, C_{ij}, and $\overline{\boldsymbol{u}}$) or by calculation (for q^n).

The `STGInflow.dat` file is a table of values at given distances from the wall. These values are then interpolated to a physical location (node or quadrature point). It has the following format:

```
[Total number of locations] 14
[d_w] [u_1] [u_2] [u_3] [R_11] [R_22] [R_33] [R_12] [R_13] [R_23] [sclr_1] [sclr_2]␣
 ↪[l_t] [eps]
```

where each `[ ]` item is a number in scientific notation (ie. `3.1415E0`), and `sclr_1` and `sclr_2` are reserved for turbulence modeling variables. They are not used in this example.

The `STGRand.dat` file is the table of the random number set, $\{\sigma^n, \boldsymbol{d}^n, \phi^n\}_{n=1}^N$. It has the format:

```
[Number of wavemodes] 7
[d_1] [d_2] [d_3] [phi] [sigma_1] [sigma_2] [sigma_3]
```

The following table is presented to help clarify the dimensionality of the numerous terms in the STG formulation.

Math	Label	$f(\boldsymbol{x})$?	$f(n)$?
$\{\sigma^n, \boldsymbol{d}^n, \phi^n\}_{n=1}^N$	RN Set	No	Yes
$\overline{\boldsymbol{u}}$	ubar	Yes	No
U_0	U0	No	No
l_t	l_t	Yes	No
ε	eps	Yes	No
$\boldsymbol{R}$	R_ij	Yes	No
$\boldsymbol{C}$	C_ij	Yes	No
q^n	q^n	Yes	Yes
$\{\kappa^n\}_{n=1}^N$	k^n	No	Yes
h_i	h_i	Yes	No
d_w	d_w	Yes	No

Internal Damping Layer (IDL)

The STG inflow boundary condition creates large amplitude acoustic waves. We use an internal damping layer (IDL) to damp them out without disrupting the synthetic structures developing into natural turbulent structures. This implementation was inspired from [SSST14], but is implemented here as a ramped volumetric forcing term, similar to a sponge layer (see 8.4.2.4 in [Col23] for example). It takes the following form:

$$S(\boldsymbol{q}) = -\sigma(\boldsymbol{x}) \left.\frac{\partial \boldsymbol{q}}{\partial \boldsymbol{Y}}\right|_{\boldsymbol{q}} \boldsymbol{Y}'$$

where $\boldsymbol{Y}' = [P - P_\text{ref}, \boldsymbol{0}, 0]^T$, and $\sigma(\boldsymbol{x})$ is a linear ramp starting at `-idl_start` with length `-idl_length` and an amplitude of inverse `-idl_decay_rate`. The damping is defined in terms of a pressure-primitive anomaly $\boldsymbol{Y}'$ converted to conservative source using $\partial \boldsymbol{q}/\partial \boldsymbol{Y}|_{\boldsymbol{q}}$, which is linearized about the current flow state. P_ref is defined via the `-reference_pressure` flag.

4.5.10.3 Meshing

The flat plate boundary layer example has custom meshing features to better resolve the flow when using a generated box mesh. These meshing features modify the nodal layout of the default, equispaced box mesh and are enabled via `-mesh_transform platemesh`. One of those is tilting the top of the domain, allowing for it to be a outflow boundary condition. The angle of this tilt is controlled by `-platemesh_top_angle`.

The primary meshing feature is the ability to grade the mesh, providing better resolution near the wall. There are two methods to do this; algorithmically, or specifying the node locations via a file. Algorithmically, a base node distribution is defined at the inlet (assumed to be $\min(x)$) and then linearly stretched/squeezed to match the slanted top boundary condition. Nodes are placed such that `-platemesh_Ndelta` elements are within `-platemesh_refine_height` of the wall. They are placed such that the element height matches a geometric growth ratio defined by `-platemesh_growth`. The remaining elements are then distributed from `-platemesh_refine_height` to the top of the domain linearly in logarithmic space.

Alternatively, a file may be specified containing the locations of each node. The file should be newline delimited, with the first line specifying the number of points and the rest being the locations of the nodes. The node locations used exactly at the inlet (assumed to be $\min(x)$) and linearly stretched/squeezed to match the slanted top boundary condition. The file is specified via `-platemesh_y_node_locs_path`. If this flag is given an empty string, then the algorithmic approach will be performed.

4.5.11 Taylor-Green Vortex

This problem is really just an initial condition, the Taylor-Green Vortex:

$$
\begin{aligned}
u &= V_0 \sin(\hat{x}) \cos(\hat{y}) \sin(\hat{z}) \\
v &= -V_0 \cos(\hat{x}) \sin(\hat{y}) \sin(\hat{z}) \\
w &= 0 \\
p &= p_0 + \frac{\rho_0 V_0^2}{16} \left(\cos(2\hat{x}) + \cos(2\hat{y}) \right) \left(\cos(2\hat{z}) + 2 \right) \\
\rho &= \frac{p}{R T_0}
\end{aligned}
$$

where $\hat{x} = 2\pi x/L$ for L the length of the domain in that specific direction. This coordinate modification is done to transform a given grid onto a domain of $x, y, z \in [0, 2\pi)$.

This initial condition is traditionally given for the incompressible Navier-Stokes equations. The reference state is selected using the `-reference_{velocity,pressure,temperature}` flags (Euclidean norm of `-reference_velocity` is used for V_0).

4.6 Solid mechanics mini-app

This example is located in the subdirectory `examples/solids`. It solves the steady-state static momentum balance equations using unstructured high-order finite/spectral element spatial discretizations. As for the *Compressible Navier-Stokes mini-app* case, the solid mechanics elasticity example has been developed using PETSc, so that the pointwise physics (defined at quadrature points) is separated from the parallelization and meshing concerns.

In this mini-app, we consider three formulations used in solid mechanics applications: linear elasticity, Neo-Hookean hyperelasticity at small strain, and Neo-Hookean hyperelasticity at finite strain. We provide the strong and weak forms of static balance of linear momentum in the small strain and finite strain regimes. The stress-strain relationship (constitutive law) for each of the material models is provided. Due to the nonlinearity of material models in Neo-Hookean hyperelasticity, the Newton linearization of the material models is provided.

Note: Linear elasticity and small-strain hyperelasticity can both by obtained from the finite-strain hyperelastic formulation by linearization of geometric and constitutive nonlinearities. The effect of these linearizations is sketched in the diagram below, where σ and ϵ are stress and strain, respectively, in the small strain regime, while $\boldsymbol{S}$ and $\boldsymbol{E}$ are their finite-strain generalizations (second Piola-Kirchoff tensor and Green-Lagrange strain tensor, respectively) defined in the initial configuration, and C is a linearized constitutive model.

$$\begin{tikzcd}
\overbrace{\boldsymbol{S}(\boldsymbol{E})}^{\text{Finite Strain Hyperelastic}} \arrow[r, "\text{constitutive}", "\text{linearization}"'] \arrow[d, "\text{geometric}"', "{\substack{\boldsymbol{E}\to\epsilon \\ \boldsymbol{S}\to\sigma}}"] & \overbrace{\boldsymbol{S} = \mathsf{C}\boldsymbol{E}}^{\text{St. Venant-Kirchoff}} \arrow[d, "{\substack{\boldsymbol{E}\to\epsilon \\ \boldsymbol{S}\to\sigma}}"', "\text{geometric}"] \\
\underbrace{\sigma(\epsilon)}_{\text{Small Strain Hyperelastic}} \arrow[r, "\text{constitutive}", "\text{linearization}"'] & \underbrace{\sigma = \mathsf{C}\epsilon}_{\text{Linear Elastic}}
\end{tikzcd} \tag{4.32}$$

4.6.1 Running the mini-app

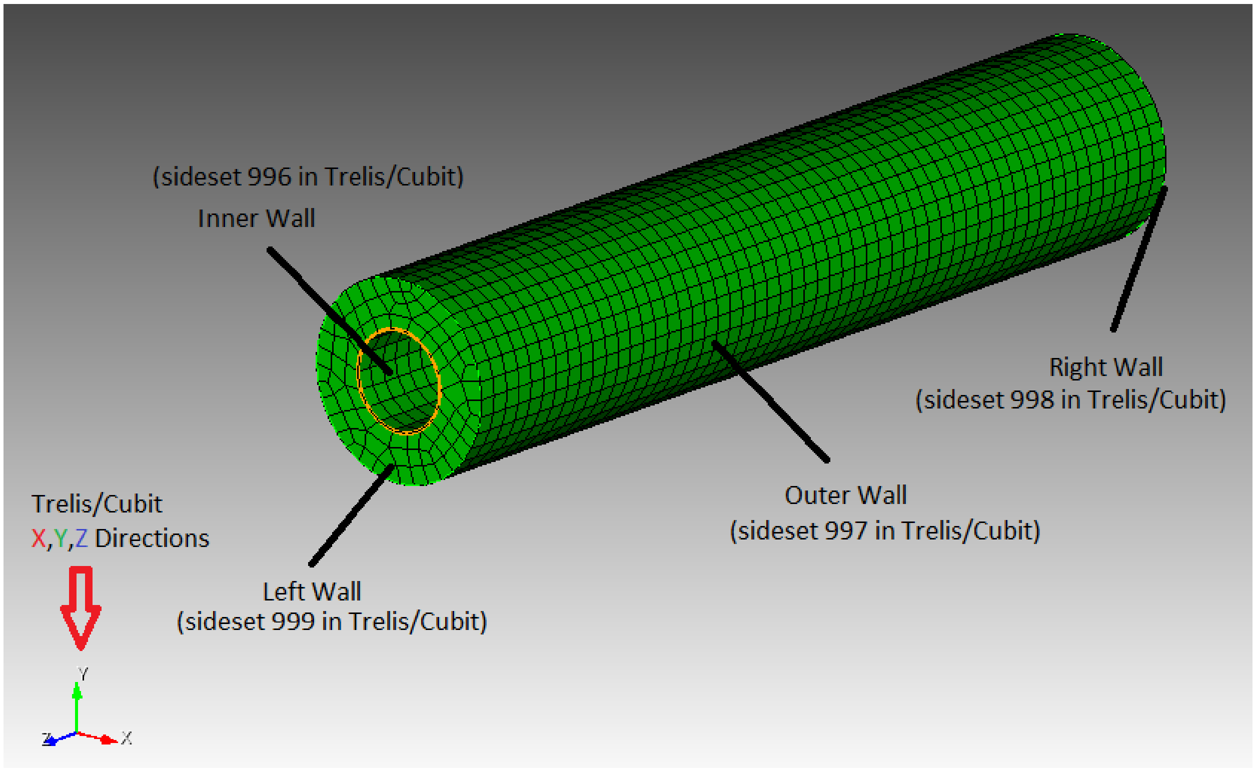
The elasticity mini-app is controlled via command-line options, the following of which are mandatory.

Table 4.15: Mandatory Runtime Options

Option	Description
`-mesh [filename]`	Path to mesh file in any format supported by PETSc.
`-degree [int]`	Polynomial degree of the finite element basis
`-E [real]`	Young's modulus, $E > 0$
`-nu [real]`	Poisson's ratio, $\nu < 0.5$
`-bc_clamp [int list]`	List of face sets on which to displace by `-bc_clamp_[facenumber]_translate [x,y,z]` and/or `bc_clamp_[facenumber]_rotate [rx,ry,rz,c_0,c_1]`. Note: The default for a clamped face is zero displacement. All displacement is with respect to the initial configuration.
`-bc_traction [int list]`	List of face sets on which to set traction boundary conditions with the traction vector `-bc_traction_[facenumber] [tx,ty,tz]`

Note: This solver can use any mesh format that PETSc's `DMPlex` can read (Exodus, Gmsh, Med, etc.). Our tests have primarily been using Exodus meshes created using CUBIT; sample meshes used for the example runs suggested here can be found in this repository. Note that many mesh formats require PETSc to be configured appropriately; e.g., `--download-exodusii` for Exodus support.

Consider the specific example of the mesh seen below:


(sideset 996 in Trelis/Cubit)
Inner Wall
Right Wall
(sideset 998 in Trelis/Cubit)
Trelis/Cubit
X,Y,Z Directions
Outer Wall
(sideset 997 in Trelis/Cubit)
Left Wall
(sideset 999 in Trelis/Cubit)
Y
Z
X

With the sidesets defined in the figure, we provide here an example of a minimal set of command line options:

```
./elasticity -mesh [.exo file] -degree 4 -E 1e6 -nu 0.3 -bc_clamp 998,999 -bc_clamp_
↪998_translate 0,-0.5,1
```

In this example, we set the left boundary, face set 999, to zero displacement and the right boundary, face set 998, to displace 0 in the x direction, -0.5 in the y, and 1 in the z.

As an alternative to specifying a mesh with `-mesh`, the user may use a DMPlex box mesh by specifying `-dm_plex_box_faces [int list]`, `-dm_plex_box_upper [real list]`, and `-dm_plex_box_lower [real list]`.

As an alternative example exploiting `-dm_plex_box_faces`, we consider a `4 x 4 x 4` mesh where essential (Drichlet) boundary condition is placed on all sides. Sides 1 through 6 are rotated around x-axis:

```
./elasticity -problem FSInitial-NH1 -E 1 -nu 0.3 -num_steps 40 -snes_linesearch_type␣
↪cp -dm_plex_box_faces 4,4,4 -bc_clamp 1,2,3,4,5,6 -bc_clamp_1_rotate 0,0,1,0,.3 -bc_
↪clamp_2_rotate 0,0,1,0,.3 -bc_clamp_3_rotate 0,0,1,0,.3 -bc_clamp_4_rotate 0,0,1,0,.
↪3 -bc_clamp_5_rotate 0,0,1,0,.3 -bc_clamp_6_rotate 0,0,1,0,.3
```

Note: If the coordinates for a particular side of a mesh are zero along the axis of rotation, it may appear that particular side is clamped zero.

On each boundary node, the rotation magnitude is computed: `theta = (c_0 + c_1 * cx) * loadIncrement` where `cx = kx * x + ky * y + kz * z`, with `kx`, `ky`, `kz` are normalized values.

The command line options just shown are the minimum requirements to run the mini-app, but additional options may also be set as follows

Table 4.16: Additional Runtime Options

Option	Description	Default value
`-ceed`	CEED resource specifier	`/cpu/self`
`-q_extra`	Number of extra quadrature points	`0`
`-test`	Run in test mode	
`-problem`	Problem to solve (`Linear`, `SS-NH`, `FSInitial-NH1`, etc.)	`Linear`
`-forcing`	Forcing term option (`none`, `constant`, or `mms`)	`none`
`-forcing_vec`	Forcing vector	`0,-1,0`
`-multigrid`	Multigrid coarsening to use (`logarithmic`, `uniform` or `none`)	`logarithmic`
`-nu_smoother [real]`	Poisson's ratio for multigrid smoothers, $\nu < 0.5$	
`-num_steps`	Number of load increments for continuation method	`1` if `Linear` else `10`
`-view_soln`	Output solution at each load increment for viewing	
`-view_final_soln`	Output solution at final load increment for viewing	
`-snes_view`	View PETSc `SNES` nonlinear solver configuration	
`-log_view`	View PETSc performance log	
`-output_dir`	Output directory	`.`
`-help`	View comprehensive information about run-time options	

To verify the convergence of the linear elasticity formulation on a given mesh with the method of manufactured solutions, run:

```
./elasticity -mesh [mesh] -degree [degree] -nu [nu] -E [E] -forcing mms
```

This option attempts to recover a known solution from an analytically computed forcing term.

4.6.1.1 On algebraic solvers

This mini-app is configured to use the following Newton-Krylov-Multigrid method by default.

- Newton-type methods for the nonlinear solve, with the hyperelasticity models globalized using load increments.
- Preconditioned conjugate gradients to solve the symmetric positive definite linear systems arising at each Newton step.
- Preconditioning via p-version multigrid coarsening to linear elements, with algebraic multigrid (PETSc's `GAMG`) for the coarse solve. The default smoother uses degree 3 Chebyshev with Jacobi preconditioning. (Lower degree is often faster, albeit less robust; try `-outer_mg_levels_ksp_max_it 2`, for example.) Application of the linear operators for all levels with degree $p > 1$ is performed matrix-free using analytic Newton linearization, while the lowest order $p = 1$ operators are assembled explicitly (using coloring at present).

Many related solvers can be implemented by composing PETSc command-line options.

4.6.1.2 Nondimensionalization

Quantities such as the Young's modulus vary over many orders of magnitude, and thus can lead to poorly scaled equations. One can nondimensionalize the model by choosing an alternate system of units, such that displacements and residuals are of reasonable scales.

Table 4.17: (Non)dimensionalization options

Option	Description	Default value
`-units_meter`	1 meter in scaled length units	1
`-units_second`	1 second in scaled time units	1
`-units_kilogram`	1 kilogram in scaled mass units	1

For example, consider a problem involving metals subject to gravity.

Table 4.18: Characteristic units for metals

Quantity	Typical value in SI units
Displacement, $\boldsymbol{u}$	$1\,\mathrm{cm} = 10^{-2}\,\mathrm{m}$
Young's modulus, E	$10^{11}\,\mathrm{Pa} = 10^{11}\,\mathrm{kg\,m^{-1}\,s^{-2}}$
Body force (gravity) on volume, $\int \rho g$	$5 \cdot 10^{4}\,\mathrm{kg\,m^{-2}\,s^{-2}} \cdot (\mathrm{volume}\,\mathrm{m}^3)$

One can choose units of displacement independently (e.g., `-units_meter 100` to measure displacement in centimeters), but E and $\int \rho g$ have the same dependence on mass and time, so cannot both be made of order 1. This reflects the fact that both quantities are not equally significant for a given displacement size; the relative significance of gravity increases as the domain size grows.

4.6.1.3 Diagnostic Quantities

Diagnostic quantities for viewing are provided when the command line options for visualization output, `-view_soln` or `-view_final_soln` are used. The diagnostic quantities include displacement in the x direction, displacement in the y direction, displacement in the z direction, pressure, trace $\boldsymbol{E}$, trace $\boldsymbol{E}^2$, $|J|$, and strain energy density. The table below summarizes the formulations of each of these quantities for each problem type.

Table 4.19: Diagnostic quantities

Quantity	Linear Elasticity	Hyperelasticity, Small Strain	Hyperelasticity, Finite Strain
Pressure	$\lambda\,\mathrm{trace}\,\boldsymbol{\epsilon}$	$\lambda \log \mathrm{trace}\,\boldsymbol{\epsilon}$	$\lambda \log J$
Volumetric Strain	$\mathrm{trace}\,\boldsymbol{\epsilon}$	$\mathrm{trace}\,\boldsymbol{\epsilon}$	$\mathrm{trace}\,\boldsymbol{E}$
trace $\boldsymbol{E}^2$	$\mathrm{trace}\,\boldsymbol{\epsilon}^2$	$\mathrm{trace}\,\boldsymbol{\epsilon}^2$	$\mathrm{trace}\,\boldsymbol{E}^2$
$\lvert J \rvert$	$1 + \mathrm{trace}\,\boldsymbol{\epsilon}$	$1 + \mathrm{trace}\,\boldsymbol{\epsilon}$	$\lvert J \rvert$
Strain Energy Density	$\frac{\lambda}{2}(\mathrm{trace}\,\boldsymbol{\epsilon})^2 + \mu\boldsymbol{\epsilon} : \boldsymbol{\epsilon}$	$\lambda(1+\mathrm{trace}\,\boldsymbol{\epsilon})(\log(1+\mathrm{trace}\,\boldsymbol{\epsilon})-1) + \mu\boldsymbol{\epsilon} : \boldsymbol{\epsilon}$	$\frac{\lambda}{2}(\log J)^2 + \mu\,\mathrm{trace}\,\boldsymbol{E} - \mu \log J$

4.6.2 Linear Elasticity

The strong form of the static balance of linear momentum at small strain for the three-dimensional linear elasticity problem is given by [Hug12]:

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{g} = \boldsymbol{0} \tag{4.33}$$

where $\boldsymbol{\sigma}$ and $\boldsymbol{g}$ are stress and forcing functions, respectively. We multiply (4.33) by a test function $\boldsymbol{v}$ and integrate the divergence term by parts to arrive at the weak form: find $\boldsymbol{u} \in \mathcal{V} \subset H^1(\Omega)$ such that

$$\int_{\Omega} \nabla \boldsymbol{v} : \boldsymbol{\sigma} \, dV - \int_{\partial\Omega} \boldsymbol{v} \cdot (\boldsymbol{\sigma} \cdot \hat{\boldsymbol{n}}) \, dS - \int_{\Omega} \boldsymbol{v} \cdot \boldsymbol{g} \, dV = 0, \quad \forall \boldsymbol{v} \in \mathcal{V}, \tag{4.34}$$

where $\boldsymbol{\sigma} \cdot \hat{\boldsymbol{n}}|_{\partial\Omega}$ is replaced by an applied force/traction boundary condition written in terms of the initial configuration. When inhomogeneous Dirichlet boundary conditions are present, $\mathcal{V}$ is an affine space that satisfies those boundary conditions.

4.6.2.1 Constitutive modeling

In their most general form, constitutive models define $\boldsymbol{\sigma}$ in terms of state variables. In the model taken into consideration in the present mini-app, the state variables are constituted by the vector displacement field $\boldsymbol{u}$, and its gradient $\nabla \boldsymbol{u}$. We begin by defining the symmetric (small/infintesimal) strain tensor as

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T\right). \tag{4.35}$$

This constitutive model $\boldsymbol{\sigma}(\boldsymbol{\epsilon})$ is a linear tensor-valued function of a tensor-valued input, but we will consider the more general nonlinear case in other models below. In these cases, an arbitrary choice of such a function will generally not be invariant under orthogonal transformations and thus will not admissible as a physical model must not depend on the coordinate system chosen to express it. In particular, given an orthogonal transformation Q, we desire

$$Q\boldsymbol{\sigma}(\boldsymbol{\epsilon})Q^T = \boldsymbol{\sigma}(Q\boldsymbol{\epsilon}Q^T), \tag{4.36}$$

which means that we can change our reference frame before or after computing $\boldsymbol{\sigma}$, and get the same result either way. Constitutive relations in which $\boldsymbol{\sigma}$ is uniquely determined by $\boldsymbol{\epsilon}$ while satisfying the invariance property (4.36) are known as Cauchy elastic materials. Here, we define a strain energy density functional $\Phi(\boldsymbol{\epsilon}) \in \mathbb{R}$ and obtain the strain energy from its gradient,

$$\boldsymbol{\sigma}(\boldsymbol{\epsilon}) = \frac{\partial \Phi}{\partial \boldsymbol{\epsilon}}. \tag{4.37}$$

Note: The strain energy density functional cannot be an arbitrary function $\Phi(\boldsymbol{\epsilon})$; it can only depend on *invariants*, scalar-valued functions γ satisfying

$$\gamma(\boldsymbol{\epsilon}) = \gamma(Q\boldsymbol{\epsilon}Q^T)$$

for all orthogonal matrices Q.

For the linear elasticity model, the strain energy density is given by

$$\boldsymbol{\Phi} = \frac{\lambda}{2}(\operatorname{trace} \boldsymbol{\epsilon})^2 + \mu \boldsymbol{\epsilon} : \boldsymbol{\epsilon}.$$

The constitutive law (stress-strain relationship) is therefore given by its gradient,

$$\boldsymbol{\sigma} = \lambda(\operatorname{trace} \boldsymbol{\epsilon})\boldsymbol{I}_3 + 2\mu\boldsymbol{\epsilon},$$

where $\boldsymbol{I}_3$ is the 3 × 3 identity matrix, the colon represents a double contraction (over both indices of $\boldsymbol{\epsilon}$), and the Lamé parameters are given by

$$\begin{aligned}\lambda &= \frac{E\nu}{(1+\nu)(1-2\nu)} \\ \mu &= \frac{E}{2(1+\nu)}\end{aligned}.$$

The constitutive law (stress-strain relationship) can also be written as

$$\boldsymbol{\sigma} = \mathsf{C}\!:\!\boldsymbol{\epsilon}. \tag{4.38}$$

For notational convenience, we express the symmetric second order tensors $\boldsymbol{\sigma}$ and $\boldsymbol{\epsilon}$ as vectors of length 6 using the Voigt notation. Hence, the fourth order elasticity tensor C (also known as elastic moduli tensor or material stiffness tensor) can be represented as

$$\mathsf{C} = \begin{pmatrix} \lambda+2\mu & \lambda & \lambda & & & \\ \lambda & \lambda+2\mu & \lambda & & & \\ \lambda & \lambda & \lambda+2\mu & & & \\ & & & \mu & & \\ & & & & \mu & \\ & & & & & \mu \end{pmatrix}. \tag{4.39}$$

Note that the incompressible limit $\nu \to \frac{1}{2}$ causes $\lambda \to \infty$, and thus C becomes singular.

4.6.3 Hyperelasticity at Small Strain

The strong and weak forms given above, in (4.33) and (4.34), are valid for Neo-Hookean hyperelasticity at small strain. However, the strain energy density differs and is given by

$$\boldsymbol{\Phi} = \lambda(1 + \operatorname{trace} \boldsymbol{\epsilon})(\log(1 + \operatorname{trace} \boldsymbol{\epsilon}) - 1) + \mu\boldsymbol{\epsilon} : \boldsymbol{\epsilon}.$$

As above, we have the corresponding constitutive law given by

$$\boldsymbol{\sigma} = \lambda \log(1 + \operatorname{trace} \boldsymbol{\epsilon})\boldsymbol{I}_3 + 2\mu\boldsymbol{\epsilon} \tag{4.40}$$

where $\boldsymbol{\epsilon}$ is defined as in (4.35).

4.6.3.1 Newton linearization

Due to nonlinearity in the constitutive law, we require a Newton linearization of (4.40). To derive the Newton linearization, we begin by expressing the derivative,

$$\mathrm{d}\boldsymbol{\sigma} = \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\epsilon}}\!:\!\mathrm{d}\boldsymbol{\epsilon}$$

where

$$\mathrm{d}\boldsymbol{\epsilon} = \frac{1}{2}\left(\nabla \mathrm{d}\boldsymbol{u} + \nabla \mathrm{d}\boldsymbol{u}^T\right)$$

and

$$\mathrm{d}\nabla \boldsymbol{u} = \nabla \,\mathrm{d}\boldsymbol{u}.$$

Therefore,

$$\mathrm{d}\boldsymbol{\sigma} = \bar{\lambda} \cdot \operatorname{trace} \mathrm{d}\boldsymbol{\epsilon} \cdot \boldsymbol{I}_3 + 2\mu \,\mathrm{d}\boldsymbol{\epsilon} \tag{4.41}$$

where we have introduced the symbol

$$\bar{\lambda} = \frac{\lambda}{1 + \epsilon_v}$$

where volumetric strain is given by $\epsilon_v = \sum_i \epsilon_{ii}$.

Equation (4.41) can be written in Voigt matrix notation as follows:

$$\begin{pmatrix} \mathrm{d}\sigma_{11} \\ \mathrm{d}\sigma_{22} \\ \mathrm{d}\sigma_{33} \\ \mathrm{d}\sigma_{23} \\ \mathrm{d}\sigma_{13} \\ \mathrm{d}\sigma_{12} \end{pmatrix} = \begin{pmatrix} 2\mu + \bar{\lambda} & \bar{\lambda} & \bar{\lambda} & & & \\ \bar{\lambda} & 2\mu + \bar{\lambda} & \bar{\lambda} & & & \\ \bar{\lambda} & \bar{\lambda} & 2\mu + \bar{\lambda} & & & \\ & & & \mu & & \\ & & & & \mu & \\ & & & & & \mu \end{pmatrix} \begin{pmatrix} \mathrm{d}\epsilon_{11} \\ \mathrm{d}\epsilon_{22} \\ \mathrm{d}\epsilon_{33} \\ 2\,\mathrm{d}\epsilon_{23} \\ 2\,\mathrm{d}\epsilon_{13} \\ 2\,\mathrm{d}\epsilon_{12} \end{pmatrix}. \tag{4.42}$$

4.6.4 Hyperelasticity at Finite Strain

In the *total Lagrangian* approach for the Neo-Hookean hyperelasticity problem, the discrete equations are formulated with respect to the initial configuration. In this formulation, we solve for displacement $\boldsymbol{u}(\boldsymbol{X})$ in the reference frame $\boldsymbol{X}$. The notation for elasticity at finite strain is inspired by [Hol00] to distinguish between the current and initial configurations. As explained in the *Common notation* section, we denote by capital letters the reference frame and by small letters the current one.

The strong form of the static balance of linear-momentum at *finite strain* (total Lagrangian) is given by:

$$-\nabla_X \cdot \boldsymbol{P} - \rho_0 \boldsymbol{g} = \boldsymbol{0} \tag{4.43}$$

where the $_X$ in ∇_X indicates that the gradient is calculated with respect to the initial configuration in the finite strain regime. $\boldsymbol{P}$ and $\boldsymbol{g}$ are the *first Piola-Kirchhoff stress* tensor and the prescribed forcing function, respectively. ρ_0 is known as the *initial* mass density. The tensor $\boldsymbol{P}$ is not symmetric, living in the current configuration on the left and the initial configuration on the right.

$\boldsymbol{P}$ can be decomposed as

$$\boldsymbol{P} = \boldsymbol{F}\,\boldsymbol{S}, \tag{4.44}$$

where $\boldsymbol{S}$ is the *second Piola-Kirchhoff stress* tensor, a symmetric tensor defined entirely in the initial configuration, and $\boldsymbol{F} = \boldsymbol{I}_3 + \nabla_X \boldsymbol{u}$ is the deformation gradient. Different constitutive models can define $\boldsymbol{S}$.

4.6.4.1 Constitutive modeling

For the constitutive modeling of hyperelasticity at finite strain, we begin by defining two symmetric tensors in the initial configuration, the right Cauchy-Green tensor

$$\boldsymbol{C} = \boldsymbol{F}^T \boldsymbol{F}$$

and the Green-Lagrange strain tensor

$$\boldsymbol{E} = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I}_3) = \frac{1}{2}\Big(\nabla_X \boldsymbol{u} + (\nabla_X \boldsymbol{u})^T + (\nabla_X \boldsymbol{u})^T \nabla_X \boldsymbol{u}\Big), \tag{4.45}$$

the latter of which converges to the linear strain tensor $\boldsymbol{\epsilon}$ in the small-deformation limit. The constitutive models considered, appropriate for large deformations, express $\boldsymbol{S}$ as a function of $\boldsymbol{E}$, similar to the linear case, shown in equation (4.38), which expresses the relationship between $\boldsymbol{\sigma}$ and $\boldsymbol{\epsilon}$.

Recall that the strain energy density functional can only depend upon invariants. We will assume without loss of generality that $\boldsymbol{E}$ is diagonal and take its set of eigenvalues as the invariants. It is clear that there can be only three invariants, and there are many alternate choices, such as $\operatorname{trace}(\boldsymbol{E})$, $\operatorname{trace}(\boldsymbol{E}^2)$, $|\boldsymbol{E}|$, and combinations thereof. It is common in the literature for invariants to be taken from $\boldsymbol{C} = \boldsymbol{I}_3 + 2\boldsymbol{E}$ instead of $\boldsymbol{E}$.

For example, if we take the compressible Neo-Hookean model,

$$\begin{aligned} \Phi(\boldsymbol{E}) &= \frac{\lambda}{2}(\log J)^2 - \mu \log J + \frac{\mu}{2}(\operatorname{trace} \boldsymbol{C} - 3) \\ &= \frac{\lambda}{2}(\log J)^2 - \mu \log J + \mu \operatorname{trace} \boldsymbol{E}, \end{aligned} \tag{4.46}$$

where $J = |\boldsymbol{F}| = \sqrt{|\boldsymbol{C}|}$ is the determinant of deformation (i.e., volume change) and λ and μ are the Lamé parameters in the infinitesimal strain limit.

To evaluate (4.37), we make use of

$$\frac{\partial J}{\partial \boldsymbol{E}} = \frac{\partial \sqrt{|\boldsymbol{C}|}}{\partial \boldsymbol{E}} = |\boldsymbol{C}|^{-1/2} |\boldsymbol{C}| \boldsymbol{C}^{-1} = J \boldsymbol{C}^{-1},$$

where the factor of $\frac{1}{2}$ has been absorbed due to $\boldsymbol{C} = \boldsymbol{I}_3 + 2\boldsymbol{E}$. Carrying through the differentiation (4.37) for the model (4.46), we arrive at

$$\boldsymbol{S} = \lambda \log J \boldsymbol{C}^{-1} + \mu(\boldsymbol{I}_3 - \boldsymbol{C}^{-1}). \tag{4.47}$$

Tip: An equivalent form of (4.47) is

$$\boldsymbol{S} = \lambda \log J \boldsymbol{C}^{-1} + 2\mu \boldsymbol{C}^{-1} \boldsymbol{E}, \tag{4.48}$$

which is more numerically stable for small $\boldsymbol{E}$, and thus preferred for computation. Note that the product $\boldsymbol{C}^{-1}\boldsymbol{E}$ is also symmetric, and that $\boldsymbol{E}$ should be computed using (4.45).

Similarly, it is preferable to compute $\log J$ using `log1p`, especially in case of nearly incompressible materials. To sketch this idea, suppose we have the 2 × 2 non-symmetric matrix $\boldsymbol{F} = \begin{pmatrix} 1+u_{0,0} & u_{0,1} \\ u_{1,0} & 1+u_{1,1} \end{pmatrix}$. Then we compute

$$\log J = \texttt{log1p}(u_{0,0} + u_{1,1} + u_{0,0}u_{1,1} - u_{0,1}u_{1,0}), \tag{4.49}$$

which gives accurate results even in the limit when the entries $u_{i,j}$ are very small. For example, if $u_{i,j} \sim 10^{-8}$, then naive computation of $\boldsymbol{I}_3 - \boldsymbol{C}^{-1}$ and $\log J$ will have a relative accuracy of order 10^{-8} in double precision and no correct digits in single precision. When using the stable choices above, these quantities retain full $\varepsilon_{\text{machine}}$ relative accuracy.

Mooney-Rivlin model

While the Neo-Hookean model depends on just two scalar invariants, $\mathbb{I}_1 = \operatorname{trace} \boldsymbol{C} = 3 + 2 \operatorname{trace} \boldsymbol{E}$ and J, Mooney-Rivlin models depend on the additional invariant, $\mathbb{I}_2 = \frac{1}{2}(\mathbb{I}_1^2 - \boldsymbol{C} : \boldsymbol{C})$. A coupled Mooney-Rivlin strain energy density (cf. Neo-Hookean (4.46)) is [Hol00]

$$\Phi(\mathbb{I}_1, \mathbb{I}_2, J) = \frac{\lambda}{2}(\log J)^2 - (\mu_1 + 2\mu_2) \log J + \frac{\mu_1}{2}(\mathbb{I}_1 - 3) + \frac{\mu_2}{2}(\mathbb{I}_2 - 3). \tag{4.50}$$

We differentiate Φ as in the Neo-Hookean case (4.47) to yield the second Piola-Kirchoff tensor,

$$\begin{aligned} \boldsymbol{S} &= \lambda \log J \boldsymbol{C}^{-1} - (\mu_1 + 2\mu_2)\boldsymbol{C}^{-1} + \mu_1 \boldsymbol{I}_3 + \mu_2(\mathbb{I}_1 \boldsymbol{I}_3 - \boldsymbol{C}) \\ &= (\lambda \log J - \mu_1 - 2\mu_2)\boldsymbol{C}^{-1} + (\mu_1 + \mu_2 \mathbb{I}_1)\boldsymbol{I}_3 - \mu_2 \boldsymbol{C}, \end{aligned} \tag{4.51}$$

where we have used

$$\frac{\partial \mathbb{I}_1}{\partial \boldsymbol{E}} = 2\boldsymbol{I}_3, \quad \frac{\partial \mathbb{I}_2}{\partial \boldsymbol{E}} = 2\mathbb{I}_1 \boldsymbol{I}_3 - 2\boldsymbol{C}, \quad \frac{\partial \log J}{\partial \boldsymbol{E}} = \boldsymbol{C}^{-1}. \tag{4.52}$$

This is a common model for vulcanized rubber, with a shear modulus (defined for the small-strain limit) of $\mu_1 + \mu_2$ that should be significantly smaller than the first Lamé parameter λ.

Mooney-Rivlin strain energy comparison

We apply traction to a block and plot integrated strain energy Φ as a function of the loading paramater.

```python
import altair as alt
import pandas as pd
def source_path(rel):
    import os
    return os.path.join(os.path.dirname(os.environ["DOCUTILSCONFIG"]), rel)

nh = pd.read_csv(source_path("examples/solids/tests-output/NH-strain.csv"))
nh["model"] = "Neo-Hookean"
nh["parameters"] = "E=2.8, nu=0.4"

mr = pd.read_csv(source_path("examples/solids/tests-output/MR-strain.csv"))
mr["model"] = "Mooney-Rivlin; Neo-Hookean equivalent"
mr["parameters"] = "mu_1=1, mu_2=0, nu=.4"

mr1 = pd.read_csv(source_path("examples/solids/tests-output/MR-strain1.csv"))
mr1["model"] = "Mooney-Rivlin"
mr1["parameters"] = "mu_1=0.5, mu_2=0.5, nu=.4"

df = pd.concat([nh, mr, mr1])
highlight = alt.selection_point(
   on = "mouseover",
   nearest = True,
   fields=["model", "parameters"],
)
base = alt.Chart(df).encode(
   alt.X("increment"),
   alt.Y("energy", scale=alt.Scale(type="sqrt")),
   alt.Color("model"),
   alt.Tooltip(("model", "parameters")),
   opacity=alt.condition(highlight, alt.value(1), alt.value(.5)),
   size=alt.condition(highlight, alt.value(2), alt.value(1)),
)
base.mark_point().add_params(highlight) + base.mark_line()
```

[graph]

Note: One can linearize (4.47) around $\boldsymbol{E} = 0$, for which $\boldsymbol{C} = \boldsymbol{I}_3 + 2\boldsymbol{E} \to \boldsymbol{I}_3$ and $J \to 1 + \operatorname{trace}\boldsymbol{E}$, therefore (4.47) reduces to

$$\boldsymbol{S} = \lambda(\operatorname{trace}\boldsymbol{E})\boldsymbol{I}_3 + 2\mu\boldsymbol{E}, \tag{4.53}$$

which is the St. Venant-Kirchoff model (constitutive linearization without geometric linearization; see (4.32)).

This model can be used for geometrically nonlinear mechanics (e.g., snap-through of thin structures), but is inappropriate for large strain.

Alternatively, one can drop geometric nonlinearities, $\boldsymbol{E} \to \boldsymbol{\epsilon}$ and $\boldsymbol{C} \to \boldsymbol{I}_3$, while retaining the nonlinear dependence on $J \to 1 + \operatorname{trace}\boldsymbol{\epsilon}$, thereby yielding (4.40) (see (4.32)).

4.6.4.2 Weak form

We multiply (4.43) by a test function $\boldsymbol{v}$ and integrate by parts to obtain the weak form for finite-strain hyperelasticity: find $\boldsymbol{u} \in \mathcal{V} \subset H^1(\Omega_0)$ such that

$$\int_{\Omega_0} \nabla_X \boldsymbol{v} : \boldsymbol{P}\, dV - \int_{\Omega_0} \boldsymbol{v} \cdot \rho_0 \boldsymbol{g}\, dV - \int_{\partial\Omega_0} \boldsymbol{v} \cdot (\boldsymbol{P} \cdot \hat{\boldsymbol{N}})\, dS = 0, \quad \forall \boldsymbol{v} \in \mathcal{V}, \tag{4.54}$$

where $\boldsymbol{P} \cdot \hat{\boldsymbol{N}}|_{\partial\Omega}$ is replaced by any prescribed force/traction boundary condition written in terms of the initial configuration. This equation contains material/constitutive nonlinearities in defining $\boldsymbol{S}(\boldsymbol{E})$, as well as geometric nonlinearities through $\boldsymbol{P} = \boldsymbol{F}\boldsymbol{S}$, $\boldsymbol{E}(\boldsymbol{F})$, and the body force $\boldsymbol{g}$, which must be pulled back from the current configuration to the initial configuration. Discretization of (4.54) produces a finite-dimensional system of nonlinear algebraic equations, which we solve using Newton-Raphson methods. One attractive feature of Galerkin discretization is that we can arrive at the same linear system by discretizing the Newton linearization of the continuous form; that is, discretization and differentiation (Newton linearization) commute.

4.6.4.3 Newton linearization

To derive a Newton linearization of (4.54), we begin by expressing the derivative of (4.44) in incremental form,

$$\mathrm{d}\boldsymbol{P} = \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}} : \mathrm{d}\boldsymbol{F} = \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \boldsymbol{F}\underbrace{\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}} : \mathrm{d}\boldsymbol{E}}_{\mathrm{d}\boldsymbol{S}} \tag{4.55}$$

where

$$\mathrm{d}\boldsymbol{E} = \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{F}} : \mathrm{d}\boldsymbol{F} = \frac{1}{2}\Big(\mathrm{d}\boldsymbol{F}^T\boldsymbol{F} + \boldsymbol{F}^T\,\mathrm{d}\boldsymbol{F}\Big)$$

and $\mathrm{d}\boldsymbol{F} = \nabla_X\,\mathrm{d}\boldsymbol{u}$. The quantity $\partial\boldsymbol{S}/\partial\boldsymbol{E}$ is known as the incremental elasticity tensor, and is analogous to the linear elasticity tensor C of (4.39). We now evaluate $\mathrm{d}\boldsymbol{S}$ for the Neo-Hookean model (4.47),

$$\mathrm{d}\boldsymbol{S} = \frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}} : \mathrm{d}\boldsymbol{E} = \lambda(\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E})\boldsymbol{C}^{-1} + 2(\mu - \lambda\log J)\boldsymbol{C}^{-1}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{C}^{-1}, \tag{4.56}$$

where we have used

$$\mathrm{d}\boldsymbol{C}^{-1} = \frac{\partial \boldsymbol{C}^{-1}}{\partial \boldsymbol{E}} : \mathrm{d}\boldsymbol{E} = -2\boldsymbol{C}^{-1}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{C}^{-1}.$$

Note: In the small-strain limit, $\boldsymbol{C} \to \boldsymbol{I}_3$ and $\log J \to 0$, thereby reducing (4.56) to the St. Venant-Kirchoff model (4.53).

Newton linearization of Mooney-Rivlin

Similar to (4.56), we differentiate (4.51) using variational notation,

$$\begin{aligned} \mathrm{d}\boldsymbol{S} = {} & \lambda(\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E})\boldsymbol{C}^{-1} \\ & + 2(\mu_1 + 2\mu_2 - \lambda \log J)\boldsymbol{C}^{-1}\,\mathrm{d}\boldsymbol{E}\boldsymbol{C}^{-1} \\ & + 2\mu_2\Big[\operatorname{trace}(\mathrm{d}\boldsymbol{E})\boldsymbol{I}_3 - \mathrm{d}\boldsymbol{E}\Big]. \end{aligned} \tag{4.57}$$

Note that this agrees with (4.56) if $\mu_1 = \mu, \mu_2 = 0$. Moving from Neo-Hookean to Mooney-Rivlin modifies the second term and adds the third.

Cancellation vs symmetry

Some cancellation is possible (at the expense of symmetry) if we substitute (4.56) into (4.55),

$$\begin{aligned} \mathrm{d}\boldsymbol{P} &= \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \lambda(\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E})\boldsymbol{F}^{-T} + 2(\mu - \lambda\log J)\boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{C}^{-1} \\ &= \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \lambda(\boldsymbol{F}^{-T} : \mathrm{d}\boldsymbol{F})\boldsymbol{F}^{-T} + (\mu - \lambda\log J)\boldsymbol{F}^{-T}(\boldsymbol{F}^{T}\,\mathrm{d}\boldsymbol{F} + \mathrm{d}\boldsymbol{F}^{T}\boldsymbol{F})\boldsymbol{C}^{-1} \\ &= \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \lambda(\boldsymbol{F}^{-T} : \mathrm{d}\boldsymbol{F})\boldsymbol{F}^{-T} + (\mu - \lambda\log J)\Big(\,\mathrm{d}\boldsymbol{F}\,\boldsymbol{C}^{-1} + \boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{F}^{T}\boldsymbol{F}^{-T}\Big), \end{aligned} \tag{4.58}$$

where we have exploited $\boldsymbol{F}\boldsymbol{C}^{-1} = \boldsymbol{F}^{-T}$ and

$$\begin{aligned} \boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E} = C^{-1}_{IJ}\,\mathrm{d}E_{IJ} &= \frac{1}{2}\boldsymbol{F}^{-1}_{Ik}\boldsymbol{F}^{-1}_{Jk}(\boldsymbol{F}_{\ell I}\,\mathrm{d}\boldsymbol{F}_{\ell J} + \mathrm{d}\boldsymbol{F}_{\ell I}\boldsymbol{F}_{\ell J}) \\ &= \frac{1}{2}\Big(\delta_{\ell k}\boldsymbol{F}^{-1}_{Jk}\,\mathrm{d}\boldsymbol{F}_{\ell J} + \delta_{\ell k}\boldsymbol{F}^{-1}_{Ik}\,\mathrm{d}\boldsymbol{F}_{\ell I}\Big) \\ &= \boldsymbol{F}^{-1}_{Ik}\,\mathrm{d}\boldsymbol{F}_{kI} = \boldsymbol{F}^{-T} : \mathrm{d}\boldsymbol{F}. \end{aligned}$$

We prefer to compute with (4.56) because (4.58) is more expensive, requiring access to (non-symmetric) $\boldsymbol{F}^{-1}$ in addition to (symmetric) $\boldsymbol{C}^{-1} = \boldsymbol{F}^{-1}\boldsymbol{F}^{-T}$, having fewer symmetries to exploit in contractions, and being less numerically stable.

$\mathrm{d}\boldsymbol{S}$ in index notation

It is sometimes useful to express (4.56) in index notation,

$$\begin{aligned} \mathrm{d}\boldsymbol{S}_{IJ} &= \frac{\partial \boldsymbol{S}_{IJ}}{\partial \boldsymbol{E}_{KL}}\,\mathrm{d}\boldsymbol{E}_{KL} \\ &= \lambda(\boldsymbol{C}^{-1}_{KL}\,\mathrm{d}\boldsymbol{E}_{KL})\boldsymbol{C}^{-1}_{IJ} + 2(\mu - \lambda\log J)\boldsymbol{C}^{-1}_{IK}\,\mathrm{d}\boldsymbol{E}_{KL}\boldsymbol{C}^{-1}_{LJ} \\ &= \underbrace{\Big(\lambda\boldsymbol{C}^{-1}_{IJ}\boldsymbol{C}^{-1}_{KL} + 2(\mu - \lambda\log J)\boldsymbol{C}^{-1}_{IK}\boldsymbol{C}^{-1}_{JL}\Big)}_{\mathsf{C}_{IJKL}}\,\mathrm{d}\boldsymbol{E}_{KL}\,, \end{aligned} \tag{4.59}$$

where we have identified the effective elasticity tensor $\mathsf{C} = \mathsf{C}_{IJKL}$. It is generally not desirable to store C, but rather to use the earlier expressions so that only 3×3 tensors (most of which are symmetric) must be manipulated. That is, given the linearization point $\boldsymbol{F}$ and solution increment $\mathrm{d}\boldsymbol{F} = \nabla_X(\mathrm{d}\boldsymbol{u})$ (which we are solving for in the Newton step), we compute $\mathrm{d}\boldsymbol{P}$ via

1. recover $\boldsymbol{C}^{-1}$ and $\log J$ (either stored at quadrature points or recomputed),
2. proceed with 3×3 matrix products as in (4.56) or the second line of (4.59) to compute $\mathrm{d}\boldsymbol{S}$ while avoiding computation or storage of higher order tensors, and
3. conclude by (4.55), where $\boldsymbol{S}$ is either stored or recomputed from its definition exactly as in the nonlinear residual evaluation.

Note that the Newton linearization of (4.54) may be written as a weak form for linear operators: find $\mathrm{d}\boldsymbol{u} \in \mathcal{V}_0$ such that

$$\int_{\Omega_0} \nabla_X \boldsymbol{v} : \mathrm{d}\boldsymbol{P} dV = \text{rhs}, \quad \forall \boldsymbol{v} \in \mathcal{V}_0,$$

where $\mathrm{d}\boldsymbol{P}$ is defined by (4.55) and (4.56), and $\mathcal{V}_0$ is the homogeneous space corresponding to $\mathcal{V}$.

Note: The decision of whether to recompute or store functions of the current state $\boldsymbol{F}$ depends on a roofline analysis [WWP09, Brown10] of the computation and the cost of the constitutive model. For low-order elements where flops tend to be in surplus relative to memory bandwidth, recomputation is likely to be preferable, where as the opposite may be true for high-order elements. Similarly, analysis with a simple constitutive model may see better performance while storing little or nothing while an expensive model such as Arruda-Boyce [AB93], which contains many special functions, may be faster when using more storage to avoid recomputation. In the case where complete linearization is preferred, note the symmetry $\mathsf{C}_{IJKL} = \mathsf{C}_{KLIJ}$ evident in (4.59), thus C can be stored as a symmetric 6×6 matrix, which has 21 unique entries. Along with 6 entries for $\boldsymbol{S}$, this totals 27 entries of overhead compared to computing everything from $\boldsymbol{F}$. This compares with 13 entries of overhead for direct storage of $\{\boldsymbol{S}, \boldsymbol{C}^{-1}, \log J\}$, which is sufficient for the Neo-Hookean model to avoid all but matrix products.

4.6.5 Hyperelasticity in current configuration

In the preceeding discussion, all equations have been formulated in the initial configuration. This may feel convenient in that the computational domain is clearly independent of the solution, but there are some advantages to defining the equations in the current configuration.

1. Body forces (like gravity), traction, and contact are more easily defined in the current configuration.
2. Mesh quality in the initial configuration can be very bad for large deformation.
3. The required storage and numerical representation can be smaller in the current configuration.

Most of the benefit in case 3 can be attained solely by moving the Jacobian representation to the current configuration [DPA+20], though residual evaluation may also be slightly faster in current configuration. There are multiple commuting paths from the nonlinear weak form in initial configuration (4.54) to the Jacobian weak form in current configuration (4.65). One may push forward to the current configuration and then linearize or linearize in initial configuration and then push forward, as summarized below.

$$\begin{tikzcd}
\overbrace{\nabla_X \boldsymbol{v} : \boldsymbol{F}\boldsymbol{S}}^{\text{Initial Residual}} \arrow[r, "\text{push forward}"] \arrow[d, "\text{linearize}"', "\substack{\mathrm{d}\boldsymbol{F} = \nabla_X \mathrm{d}\boldsymbol{u} \\ \mathrm{d}\boldsymbol{S}(\mathrm{d}\boldsymbol{E})}"] & \overbrace{\nabla_x \boldsymbol{v} : \boldsymbol{\tau}}^{\text{Current Residual}} \arrow[d, "\substack{\mathrm{d}\nabla_x \boldsymbol{v} = -\nabla_x \boldsymbol{v} \nabla_x \mathrm{d}\boldsymbol{u} \\ \mathrm{d}\boldsymbol{\tau}(\mathrm{d}\boldsymbol{\epsilon})}"', "\text{linearize}"] \\
\underbrace{\nabla_X \boldsymbol{v} : \Big(\mathrm{d}\boldsymbol{F}\boldsymbol{S} + \boldsymbol{F}\mathrm{d}\boldsymbol{S} \Big)}_{\text{Initial Jacobian}} \arrow[r, "\text{push forward}"] & \underbrace{\nabla_x \boldsymbol{v} : \Big(\mathrm{d}\boldsymbol{\tau} - \boldsymbol{\tau}(\nabla_x \mathrm{d}\boldsymbol{u})^T \Big)}_{\text{Current Jacobian}}
\end{tikzcd} \tag{4.60}$$

We will follow both paths for consistency and because both intermediate representations may be useful for implementation.

4.6.5.1 Push forward, then linearize

The first term of (4.54) can be rewritten in terms of the symmetric Kirchhoff stress tensor $\boldsymbol{\tau} = J\sigma = \boldsymbol{P}\boldsymbol{F}^T = \boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T$ as

$$\nabla_X \boldsymbol{v} : \boldsymbol{P} = \nabla_X \boldsymbol{v} : \boldsymbol{\tau}\boldsymbol{F}^{-T} = \nabla_X \boldsymbol{v}\boldsymbol{F}^{-1} : \boldsymbol{\tau} = \nabla_x \boldsymbol{v} : \boldsymbol{\tau}$$

therefore, the weak form in terms of $\boldsymbol{\tau}$ and ∇_x with integral over Ω_0 is

$$\int_{\Omega_0} \nabla_x \boldsymbol{v} : \boldsymbol{\tau} \, dV - \int_{\Omega_0} \boldsymbol{v} \cdot \rho_0 \boldsymbol{g} \, dV - \int_{\partial\Omega_0} \boldsymbol{v} \cdot (\boldsymbol{P} \cdot \hat{\boldsymbol{N}}) \, dS = 0, \quad \forall \boldsymbol{v} \in \mathcal{V}. \tag{4.61}$$

Linearize in current configuration

To derive a Newton linearization of (4.61), first we define

$$\nabla_x \, \mathrm{d}\boldsymbol{u} = \nabla_X \, \mathrm{d}\boldsymbol{u} \, \boldsymbol{F}^{-1} = \mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1} \tag{4.62}$$

and $\boldsymbol{\tau}$ for Neo-Hookean materials as the push forward of (4.47)

$$\boldsymbol{\tau} = \boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T = \mu(\boldsymbol{b} - \boldsymbol{I}_3) + \lambda \log J \boldsymbol{I}_3, \tag{4.63}$$

where $\boldsymbol{b} = \boldsymbol{F}\boldsymbol{F}^T$, is the left Cauchy-Green tensor. Then by expanding the directional derivative of $\nabla_x \boldsymbol{v} : \boldsymbol{\tau}$, we arrive at

$$\mathrm{d}\left(\nabla_x \boldsymbol{v} : \boldsymbol{\tau}\right) = \mathrm{d}\left(\nabla_x \boldsymbol{v}\right) : \boldsymbol{\tau} + \nabla_x \boldsymbol{v} : \mathrm{d}\boldsymbol{\tau}. \tag{4.64}$$

The first term of (4.64) can be written as

$$\begin{aligned}
\mathrm{d}\left(\nabla_x \boldsymbol{v}\right) : \boldsymbol{\tau} = \mathrm{d}\left(\nabla_X \boldsymbol{v}\boldsymbol{F}^{-1}\right) : \boldsymbol{\tau} &= \Big(\underbrace{\nabla_X (\mathrm{d}\boldsymbol{v})}_{0} \boldsymbol{F}^{-1} + \nabla_X \boldsymbol{v} \, \mathrm{d}\boldsymbol{F}^{-1} \Big) : \boldsymbol{\tau} \\
&= \Big(-\nabla_X \boldsymbol{v}\boldsymbol{F}^{-1} \, \mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1} \Big) : \boldsymbol{\tau} = \Big(-\nabla_x \boldsymbol{v} \, \mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1} \Big) : \boldsymbol{\tau} \\
&= \Big(-\nabla_x \boldsymbol{v} \nabla_x \, \mathrm{d}\boldsymbol{u} \Big) : \boldsymbol{\tau} = -\nabla_x \boldsymbol{v} : \boldsymbol{\tau} (\nabla_x \, \mathrm{d}\boldsymbol{u})^T ,
\end{aligned}$$

where we have used $\mathrm{d}\boldsymbol{F}^{-1} = -\boldsymbol{F}^{-1} \, \mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1}$ and (4.62). Using this and (4.64) in (4.61) yields the weak form in the current configuration

$$\int_{\Omega_0} \nabla_x \boldsymbol{v} : \Big(\mathrm{d}\boldsymbol{\tau} - \boldsymbol{\tau} (\nabla_x \, \mathrm{d}\boldsymbol{u})^T \Big) = \text{rhs}. \tag{4.65}$$

In the following, we will sometimes make use of the incremental strain tensor in the current configuration,

$$\mathrm{d}\boldsymbol{\epsilon} \equiv \frac{1}{2} \Big(\nabla_x \, \mathrm{d}\boldsymbol{u} + (\nabla_x \, \mathrm{d}\boldsymbol{u})^T \Big).$$

Deriving $\mathrm{d}\tau$ for Neo-Hookean material

To derive a useful expression of $\mathrm{d}\boldsymbol{\tau}$ for Neo-Hookean materials, we will use the representations

$$\begin{aligned}
\mathrm{d}\boldsymbol{b} &= \mathrm{d}\boldsymbol{F}\boldsymbol{F}^T + \boldsymbol{F} \, \mathrm{d}\boldsymbol{F}^T \\
&= \nabla_x \, \mathrm{d}\boldsymbol{u} \, \boldsymbol{b} + \boldsymbol{b} \, (\nabla_x \, \mathrm{d}\boldsymbol{u})^T \\
&= (\nabla_x \, \mathrm{d}\boldsymbol{u})(\boldsymbol{b} - \boldsymbol{I}_3) + (\boldsymbol{b} - \boldsymbol{I}_3)(\nabla_x \, \mathrm{d}\boldsymbol{u})^T + 2 \, \mathrm{d}\boldsymbol{\epsilon}
\end{aligned}$$

and

$$\begin{aligned}
\mathrm{d}\,(\log J) &= \frac{\partial \log J}{\partial \boldsymbol{b}} : \mathrm{d}\boldsymbol{b} = \frac{\partial J}{J \partial \boldsymbol{b}} : \mathrm{d}\boldsymbol{b} = \frac{1}{2}\boldsymbol{b}^{-1} : \mathrm{d}\boldsymbol{b} \\
&= \frac{1}{2}\boldsymbol{b}^{-1} : \Big(\nabla_x \, \mathrm{d}\boldsymbol{u}\, \boldsymbol{b} + \boldsymbol{b}(\nabla_x \, \mathrm{d}\boldsymbol{u})^T\Big) \\
&= \operatorname{trace}(\nabla_x \, \mathrm{d}\boldsymbol{u}) \\
&= \operatorname{trace} \mathrm{d}\boldsymbol{\epsilon}.
\end{aligned}$$

Substituting into (4.63) gives

$$\begin{aligned}
\mathrm{d}\boldsymbol{\tau} &= \mu \, \mathrm{d}\boldsymbol{b} + \lambda \operatorname{trace}(\mathrm{d}\boldsymbol{\epsilon})\boldsymbol{I}_3 \\
&= \underbrace{2\mu \, \mathrm{d}\boldsymbol{\epsilon} + \lambda \operatorname{trace}(\mathrm{d}\boldsymbol{\epsilon})\boldsymbol{I}_3 - 2\lambda \log J \, \mathrm{d}\boldsymbol{\epsilon}}_{\boldsymbol{F}\,\mathrm{d}\boldsymbol{S}\boldsymbol{F}^T} \\
&\quad + (\nabla_x \, \mathrm{d}\boldsymbol{u}) \underbrace{\Big(\mu(\boldsymbol{b} - \boldsymbol{I}_3) + \lambda \log J \boldsymbol{I}_3\Big)}_{\boldsymbol{\tau}} \\
&\quad + \underbrace{\Big(\mu(\boldsymbol{b} - \boldsymbol{I}_3) + \lambda \log J \boldsymbol{I}_3\Big)}_{\boldsymbol{\tau}} (\nabla_x \, \mathrm{d}\boldsymbol{u})^T,
\end{aligned} \tag{4.66}$$

where the final expression has been identified according to

$$\mathrm{d}\boldsymbol{\tau} = \mathrm{d}\,(\boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T) = (\nabla_x \, \mathrm{d}\boldsymbol{u})\boldsymbol{\tau} + \boldsymbol{F}\,\mathrm{d}\boldsymbol{S}\boldsymbol{F}^T + \boldsymbol{\tau}(\nabla_x \, \mathrm{d}\boldsymbol{u})^T.$$

Collecting terms, we may thus opt to use either of the two forms

$$\begin{aligned}
\mathrm{d}\boldsymbol{\tau} - \boldsymbol{\tau}(\nabla_x \, \mathrm{d}\boldsymbol{u})^T &= (\nabla_x \, \mathrm{d}\boldsymbol{u})\boldsymbol{\tau} + \boldsymbol{F}\,\mathrm{d}\boldsymbol{S}\boldsymbol{F}^T \\
&= (\nabla_x \, \mathrm{d}\boldsymbol{u})\boldsymbol{\tau} + \lambda \operatorname{trace}(\mathrm{d}\boldsymbol{\epsilon})\boldsymbol{I}_3 + 2(\mu - \lambda \log J)\, \mathrm{d}\boldsymbol{\epsilon},
\end{aligned} \tag{4.67}$$

with the last line showing the especially compact representation available for Neo-Hookean materials.

4.6.5.2 Linearize, then push forward

We can move the derivatives to the current configuration via

$$\nabla_X \boldsymbol{v} : \mathrm{d}\boldsymbol{P} = (\nabla_X \boldsymbol{v})\boldsymbol{F}^{-1} : \mathrm{d}\boldsymbol{P}\boldsymbol{F}^T = \nabla_x \boldsymbol{v} : \mathrm{d}\boldsymbol{P}\boldsymbol{F}^T$$

and expand

$$\begin{aligned}
\mathrm{d}\boldsymbol{P}\boldsymbol{F}^T &= \mathrm{d}\boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T + \boldsymbol{F}\,\mathrm{d}\boldsymbol{S}\boldsymbol{F}^T \\
&= \underbrace{\mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1}}_{\nabla_x \mathrm{d}\boldsymbol{u}} \underbrace{\boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T}_{\boldsymbol{\tau}} + \boldsymbol{F}\,\mathrm{d}\boldsymbol{S}\boldsymbol{F}^T.
\end{aligned}$$

Representation of $F \, \mathrm{d}SF^T$ for Neo-Hookean materials

Now we push (4.56) forward via

$$\begin{aligned}
\boldsymbol{F}\,\mathrm{d}\boldsymbol{S}\boldsymbol{F}^T &= \lambda(\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E})\boldsymbol{F}\boldsymbol{C}^{-1}\boldsymbol{F}^T + 2(\mu - \lambda \log J)\boldsymbol{F}\boldsymbol{C}^{-1}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{C}^{-1}\boldsymbol{F}^T \\
&= \lambda(\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E})\boldsymbol{I}_3 + 2(\mu - \lambda \log J)\boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{F}^{-1} \\
&= \lambda \operatorname{trace}(\nabla_x \, \mathrm{d}\boldsymbol{u})\boldsymbol{I}_3 + 2(\mu - \lambda \log J)\, \mathrm{d}\boldsymbol{\epsilon}
\end{aligned}$$

where we have used

$$\begin{aligned}
\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E} &= \boldsymbol{F}^{-1}\boldsymbol{F}^{-T} : \boldsymbol{F}^{T}\,\mathrm{d}\boldsymbol{F} \\
&= \operatorname{trace}(\boldsymbol{F}^{-1}\boldsymbol{F}^{-T}\boldsymbol{F}^{T}\,\mathrm{d}\boldsymbol{F}) \\
&= \operatorname{trace}(\boldsymbol{F}^{-1}\,\mathrm{d}\boldsymbol{F}) \\
&= \operatorname{trace}(\mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1}) \\
&= \operatorname{trace}(\nabla_x\,\mathrm{d}\boldsymbol{u})
\end{aligned}$$

and

$$\begin{aligned}
\boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{F}^{-1} &= \frac{1}{2}\boldsymbol{F}^{-T}(\boldsymbol{F}^{T}\,\mathrm{d}\boldsymbol{F} + \mathrm{d}\boldsymbol{F}^{T}\boldsymbol{F})\boldsymbol{F}^{-1} \\
&= \frac{1}{2}(\mathrm{d}\boldsymbol{F}\boldsymbol{F}^{-1} + \boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{F}^{T}) \\
&= \frac{1}{2}\Big(\nabla_x\,\mathrm{d}\boldsymbol{u} + (\nabla_x\,\mathrm{d}\boldsymbol{u})^{T}\Big) \equiv \mathrm{d}\boldsymbol{\epsilon}.
\end{aligned}$$

Collecting terms, the weak form of the Newton linearization for Neo-Hookean materials in the current configuration is

$$\int_{\Omega_0} \nabla_x \boldsymbol{v} : \Big((\nabla_x\,\mathrm{d}\boldsymbol{u})\boldsymbol{\tau} + \lambda\operatorname{trace}(\mathrm{d}\boldsymbol{\epsilon})\boldsymbol{I}_3 + 2(\mu - \lambda\log J)\,\mathrm{d}\boldsymbol{\epsilon}\Big) dV = \text{rhs}, \tag{4.68}$$

which equivalent to Algorithm 2 of [DPA+20] and requires only derivatives with respect to the current configuration. Note that (4.67) and (4.68) have recovered the same representation using different algebraic manipulations.

Tip: We define a second order *Green-Euler* strain tensor (cf. Green-Lagrange strain (4.45)) as

$$\boldsymbol{e} = \frac{1}{2}\Big(\boldsymbol{b} - \boldsymbol{I}_3\Big) = \frac{1}{2}\Big(\nabla_X \boldsymbol{u} + (\nabla_X \boldsymbol{u})^{T} + \nabla_X \boldsymbol{u}\,(\nabla_X \boldsymbol{u})^{T}\Big). \tag{4.69}$$

Then, the Kirchhoff stress tensor (4.63) can be written as

$$\boldsymbol{\tau} = \lambda \log J \boldsymbol{I}_3 + 2\mu \boldsymbol{e}, \tag{4.70}$$

which is more numerically stable for small strain, and thus preferred for computation. Note that the $\log J$ is computed via `log1p` (4.49), as we discussed in the previous tip.

4.6.5.3 Jacobian representation

We have implemented four storage variants for the Jacobian in our finite strain hyperelasticity. In each case, some variables are computed during residual evaluation and used during Jacobian application.

Table 4.20: Four algorithms for Jacobian action in finite strain hyperelasticity problem

Option `-problem`	Static storage	Computed storage	# scalars	Equations
`FSInitial-NH1`	$\nabla_X \hat{X}, \det \nabla_{\hat{X}} X$	$\nabla_X \boldsymbol{u}$	19	(4.55) (4.56)
`FSInitial-NH2`	$\nabla_X \hat{X}, \det \nabla_{\hat{X}} X$	$\nabla_X \boldsymbol{u}, \boldsymbol{C}^{-1}, \lambda \log J$	26	(4.55) (4.56)
`FSCurrent-NH1`	$\nabla_X \hat{X}, \det \nabla_{\hat{X}} X$	$\nabla_X \boldsymbol{u}$	19	(4.65) (4.62)
`FSCurrent-NH2`	$\det \nabla_{\hat{X}} X$	$\nabla_x \hat{X}, \boldsymbol{\tau}, \lambda \log J$	17	(4.65) (4.68)

5 Julia, Python, and Rust Interfaces

libCEED provides high-level interfaces using the Julia, Python, and Rust programming languages.

More information about the Julia interface can be found at the LibCEED.jl documentation.

Usage of the Python interface is illustrated through a sequence of Jupyter Notebook tutorials. More information on the Python interface is available in the SciPy paper.

More information about the Rust interface can be found at the Rust interface documentation.

6 API Documentation

This section contains the code documentation. The subsections represent the different API objects, typedefs, and enumerations.

6.1 Public API

These objects and functions are intended to be used by general users of libCEED and can generally be found in *ceed.h*.

6.1.1 Ceed

A *Ceed* is a library context representing control of a logical hardware resource.

6.1.1.1 Base library resources

typedef struct Ceed_private ***Ceed**

Typedefs and macros used in public interfaces and user QFunction source.

This line prevents IWYU from suggesting “ceed.h” Library context created by *CeedInit*()

typedef struct CeedRequest_private ***CeedRequest**

Non-blocking Ceed interfaces return a CeedRequest.

To perform an operation immediately, pass CEED_REQUEST_IMMEDIATE instead.

CeedRequest *const **CEED_REQUEST_IMMEDIATE** = &ceed_request_immediate

Request immediate completion.

This predefined constant is passed as the *CeedRequest* argument to interfaces when the caller wishes for the operation to be performed immediately. The code

```
CeedOperatorApply(op, ..., CEED_REQUEST_IMMEDIATE);
```

is semantically equivalent to

```
CeedRequest request;
CeedOperatorApply(op, ..., &request);
CeedRequestWait(&request);
```

See also:

CEED_REQUEST_ORDERED

CeedRequest *const **CEED_REQUEST_ORDERED** = &ceed_request_ordered

Request ordered completion.

This predefined constant is passed as the *CeedRequest* argument to interfaces when the caller wishes for the operation to be completed in the order that it is submitted to the device. It is typically used in a construct such as:

```
CeedRequest request;
CeedOperatorApply(op1, ..., CEED_REQUEST_ORDERED);
CeedOperatorApply(op2, ..., &request);
// other optional work
CeedRequestWait(&request);
```

which allows the sequence to complete asynchronously but does not start `op2` until `op1` has completed.

Todo:

The current implementation is overly strict, offering equivalent semantics to CEED_REQUEST_IMMEDIATE.

See also:

CEED_REQUEST_IMMEDIATE

int **CeedRequestWait**(*CeedRequest* *req)

Wait for a CeedRequest to complete.

Calling CeedRequestWait on a NULL request is a no-op.

User Functions

Parameters

- **req** – Address of CeedRequest to wait for; zeroed on completion.

Returns

An error code: 0 - success, otherwise - failure

int **CeedRegistryGetList**(size_t *n, char ***const resources, *CeedInt* **priorities)

Get the list of available resource names for Ceed contexts.

Note: The caller is responsible for `free()`ing the resources and priorities arrays, but should not `free()` the contents of the resources array.

User Functions

Parameters

- **n** – [**out**] Number of available resources
- **resources** – [**out**] List of available resource names
- **priorities** – [**out**] Resource name prioritization values, lower is better

Returns

An error code: 0 - success, otherwise - failure

int **CeedInit**(const char *resource, *Ceed* *ceed)

Initialize a Ceed: core components context to use the specified resource.

Note: Prefixing the resource with "help:" (e.g. "help:/cpu/self") will result in CeedInt printing the current libCEED version number and a list of current available backend resources to stderr.

User Functions

See also:

CeedRegister() *CeedDestroy()*

Parameters

- **`resource`** – [**in**] Resource to use, e.g., "/cpu/self"
- **`ceed`** – [**out**] The library context

Returns

An error code: 0 - success, otherwise - failure

int **CeedSetStream**(*Ceed* ceed, void *handle)

Set the GPU stream for a Ceed context.

User Functions

Parameters

- **`ceed`** – [**inout**] Ceed context to set the stream
- **`handle`** – [**in**] Handle to GPU stream

Returns

An error code: 0 - success, otherwise - failure

int **CeedReferenceCopy**(*Ceed* ceed, *Ceed* *ceed_copy)

Copy the pointer to a Ceed context.

Both pointers should be destroyed with `CeedDestroy()`.

Note: If the value of `ceed_copy` passed to this function is non-NULL, then it is assumed that `ceed_copy` is a pointer to a Ceed context. This Ceed context will be destroyed if `ceed_copy` is the only reference to this Ceed context.

User Functions

Parameters

- **`ceed`** – [**in**] Ceed context to copy reference to
- **`ceed_copy`** – [**inout**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **CeedGetResource**(*Ceed* ceed, const char **resource)

Get the full resource name for a Ceed context.

User Functions

Parameters

- **`ceed`** – [**in**] Ceed context to get resource name of
- **`resource`** – [**out**] Variable to store resource name

Returns

An error code: 0 - success, otherwise - failure

int **CeedGetPreferredMemType**(*Ceed* ceed, *CeedMemType* *mem_type)

Return Ceed context preferred memory type.

User Functions

Parameters

- **ceed** – [**in**] Ceed context to get preferred memory type of
- **mem_type** – [**out**] Address to save preferred memory type to

Returns

An error code: 0 - success, otherwise - failure

int **CeedIsDeterministic**(*Ceed* ceed, bool *is_deterministic)

Get deterministic status of Ceed.

User Functions

Parameters

- **ceed** – [**in**] Ceed
- **is_deterministic** – [**out**] Variable to store deterministic status

Returns

An error code: 0 - success, otherwise - failure

int **CeedAddJitSourceRoot**(*Ceed* ceed, const char *jit_source_root)

Set additional JiT source root for Ceed.

User Functions

Parameters

- **ceed** – [**inout**] Ceed
- **jit_source_root** – [**in**] Absolute path to additional JiT source directory

Returns

An error code: 0 - success, otherwise - failure

int **CeedView**(*Ceed* ceed, FILE *stream)

View a Ceed.

User Functions

Parameters

- **ceed** – [**in**] Ceed to view
- **stream** – [**in**] Filestream to write to

Returns

An error code: 0 - success, otherwise - failure

int **CeedDestroy**(*Ceed* *ceed)

Destroy a Ceed context.

User Functions

Parameters

- **ceed** – [**inout**] Address of Ceed context to destroy

Returns

An error code: 0 - success, otherwise - failure

const char ***CeedErrorFormat**(*Ceed* ceed, const char *format, va_list *args)

int **CeedErrorImpl**(*Ceed* ceed, const char *filename, int lineno, const char *func, int ecode, const char *format, ...)

Error handling implementation; use *CeedError* instead.

Library Developer Functions

int **CeedErrorReturn**(*Ceed* ceed, const char *filename, int line_no, const char *func, int err_code, const char *format, va_list *args)

Error handler that returns without printing anything.

Ceed error handlers.

Pass this to *CeedSetErrorHandler()* to obtain this error handling behavior.

Library Developer Functions

int **CeedErrorStore**(*Ceed* ceed, const char *filename, int line_no, const char *func, int err_code, const char *format, va_list *args)

Error handler that stores the error message for future use and returns the error.

Pass this to *CeedSetErrorHandler()* to obtain this error handling behavior.

Library Developer Functions

int **CeedErrorAbort**(*Ceed* ceed, const char *filename, int line_no, const char *func, int err_code, const char *format, va_list *args)

Error handler that prints to stderr and aborts.

Pass this to *CeedSetErrorHandler()* to obtain this error handling behavior.

Library Developer Functions

int **CeedErrorExit**(*Ceed* ceed, const char *filename, int line_no, const char *func, int err_code, const char *format, va_list *args)

Error handler that prints to stderr and exits.

Pass this to *CeedSetErrorHandler()* to obtain this error handling behavior.

In contrast to *CeedErrorAbort()*, this exits without a signal, so atexit() handlers (e.g., as used by gcov) are run.

Library Developer Functions

int **CeedSetErrorHandler**(*Ceed* ceed, CeedErrorHandler handler)

Set error handler.

A default error handler is set in *CeedInit()*. Use this function to change the error handler to *CeedErrorReturn()*, *CeedErrorAbort()*, or a user-defined error handler.

Library Developer Functions

int **CeedGetErrorMessage**(*Ceed* ceed, const char **err_msg)

Get error message.

The error message is only stored when using the error handler *CeedErrorStore()*

Library Developer Functions

Parameters

- **ceed** – [**in**] Ceed context to retrieve error message
- **err_msg** – [**out**] Char pointer to hold error message

int **CeedResetErrorMessage**(*Ceed* ceed, const char **err_msg)

Restore error message.

The error message is only stored when using the error handler *CeedErrorStore*()

Library Developer Functions

Parameters

- **ceed** – [**in**] Ceed context to restore error message
- **err_msg** – [**out**] Char pointer that holds error message

int **CeedGetVersion**(int *major, int *minor, int *patch, bool *release)

Get libCEED library version info.

libCEED version numbers have the form major.minor.patch. Non-release versions may contain unstable interfaces.

The caller may pass NULL for any arguments that are not needed.

Library Developer Functions

See also:

CEED_VERSION_GE()

Parameters

- **major** – [**out**] Major version of the library
- **minor** – [**out**] Minor version of the library
- **patch** – [**out**] Patch (subminor) version of the library
- **release** – [**out**] True for releases; false for development branches

int **CeedGetScalarType**(CeedScalarType *scalar_type)

Get libCEED scalar type, such as F64 or F32.

Library Developer Functions

Parameters

- **scalar_type** – [**out**] Type of libCEED scalars

Macros

CeedError(ceed, ecode, ...)

Raise an error on ceed object.

See also:

CeedSetErrorHandler()

Parameters

- **ceed** – Ceed library context or NULL
- **ecode** – Error code (int)

- **...** – printf-style format string followed by arguments as needed

CeedPragmaSIMD

This macro provides the appropriate SIMD Pragma for the compilation environment.

Code generation backends may redefine this macro, as needed.

CEED_VERSION_GE(major, minor, patch)

Compile-time check that the the current library version is at least as recent as the specified version.

This macro is typically used in

```
#if CEED_VERSION_GE(0, 8, 0)
  code path that needs at least 0.8.0
#else
  fallback code for older versions
#endif
```

A non-release version always compares as positive infinity.

See also:

CeedGetVersion()

Parameters

- **major** – Major version
- **minor** – Minor version
- **patch** – Patch (subminor) version

Typedefs and Enumerations

enum **CeedMemType**

Specify memory type.

Many Ceed interfaces take or return pointers to memory. This enum is used to specify where the memory being provided or requested must reside.

Values:

enumerator **CEED_MEM_HOST**

Memory resides on the host.

enumerator **CEED_MEM_DEVICE**

Memory resides on a device (corresponding to Ceed: core components resource)

enum **CeedErrorType**

Base scalar type for the library to use: change which header is included to change the precision.

Ceed error code.

This enum is used to specify the type of error returned by a function. A zero error code is success, negative error codes indicate terminal errors and positive error codes indicate nonterminal errors.

With nonterminal errors the object state has not been modified, but with terminal errors the object data is likely modified or corrupted.

Values:

enumerator **CEED_ERROR_SUCCESS**

Success error code.

enumerator **CEED_ERROR_MINOR**

Minor error, generic.

enumerator **CEED_ERROR_DIMENSION**

Minor error, dimension mismatch in inputs.

enumerator **CEED_ERROR_INCOMPLETE**

Minor error, incomplete object setup.

enumerator **CEED_ERROR_INCOMPATIBLE**

Minor error, incompatible arguments/configuration.

enumerator **CEED_ERROR_ACCESS**

Minor error, access lock problem.

enumerator **CEED_ERROR_MAJOR**

Major error, generic.

enumerator **CEED_ERROR_BACKEND**

Major error, internal backend error.

enumerator **CEED_ERROR_UNSUPPORTED**

Major error, operation unsupported by current backend.

6.1.2 CeedVector

A *CeedVector* constitutes the main data structure and serves as input/output for the *CeedOperator*s.

6.1.2.1 Basic vector operations

typedef struct CeedVector_private ***CeedVector**

Handle for vectors over the field *CeedScalar*.

const *CeedVector* **CEED_VECTOR_ACTIVE** = &ceed_vector_active

Indicate that vector will be provided as an explicit argument to *CeedOperatorApply()*.

const *CeedVector* **CEED_VECTOR_NONE** = &ceed_vector_none

Indicate that no vector is applicable (i.e., for *CEED_EVAL_WEIGHT*).

int **`CeedVectorCreate`**(*Ceed* ceed, CeedSize length, *CeedVector* *vec)

Create a CeedVector of the specified length (does not allocate memory)

User Functions

Parameters

- **`ceed`** – [**in**] Ceed object where the CeedVector will be created
- **`length`** – [**in**] Length of vector
- **`vec`** – [**out**] Address of the variable where the newly created CeedVector will be stored

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorReferenceCopy`**(*CeedVector* vec, *CeedVector* *vec_copy)

Copy the pointer to a CeedVector.

Both pointers should be destroyed with *`CeedVectorDestroy()`*.

Note: If the value of `vec_copy` passed to this function is non-NULL, then it is assumed that `vec_copy` is a pointer to a CeedVector. This CeedVector will be destroyed if `vec_copy` is the only reference to this CeedVector.

User Functions

Parameters

- **`vec`** – [**in**] CeedVector to copy reference to
- **`vec_copy`** – [**inout**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorCopy`**(*CeedVector* vec, *CeedVector* vec_copy)

Copy a CeedVector into a different CeedVector.

Both pointers should be destroyed with *`CeedVectorDestroy()`*.

Note: If `*vec_copy` is non-NULL, then it is assumed that `*vec_copy` is a pointer to a CeedVector. This CeedVector will be destroyed if `*vec_copy` is the only reference to this CeedVector.

User Functions

Parameters

- **`vec`** – [**in**] CeedVector to copy
- **`vec_copy`** – [**inout**] Variable to store copied CeedVector to

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorSetArray`**(*CeedVector* vec, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, *CeedScalar* *array)

Set the array used by a CeedVector, freeing any previously allocated array if applicable.

The backend may copy values to a different memtype, such as during *CeedOperatorApply()*. See also *CeedVectorSyncArray()* and *CeedVectorTakeArray()*.

User Functions

Parameters

- **vec** – [**inout**] CeedVector
- **mem_type** – [**in**] Memory type of the array being passed
- **copy_mode** – [**in**] Copy mode for the array
- **array** – [**in**] Array to be used, or NULL with *CEED_COPY_VALUES* to have the library allocate

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorSetValue**(*CeedVector* vec, *CeedScalar* value)

Set the CeedVector to a constant value.

User Functions

Parameters

- **vec** – [**inout**] CeedVector
- **value** – [**in**] Value to be used

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorSyncArray**(*CeedVector* vec, *CeedMemType* mem_type)

Sync the CeedVector to a specified memtype.

This function is used to force synchronization of arrays set with *CeedVectorSetArray()*. If the requested memtype is already synchronized, this function results in a no-op.

User Functions

Parameters

- **vec** – [**inout**] CeedVector
- **mem_type** – [**in**] Memtype to be synced

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorTakeArray**(*CeedVector* vec, *CeedMemType* mem_type, *CeedScalar* **array)

Take ownership of the CeedVector array set by *CeedVectorSetArray()* with *CEED_USE_POINTER* and remove the array from the CeedVector.

The caller is responsible for managing and freeing the array. This function will error if *CeedVectorSetArray()* was not previously called with *CEED_USE_POINTER* for the corresponding mem_type.

User Functions

Parameters

- **vec** – [**inout**] CeedVector
- **mem_type** – [**in**] Memory type on which to take the array. If the backend uses a different memory type, this will perform a copy.
- **array** – [**out**] Array on memory type mem_type, or NULL if array pointer is not required

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorGetArray**(*CeedVector* vec, *CeedMemType* mem_type, *CeedScalar* **array)

Get read/write access to a CeedVector via the specified memory type.

Restore access with *CeedVectorRestoreArray()*.

User Functions

Note: The CeedVectorGetArray* and CeedVectorRestoreArray* functions provide access to array pointers in the desired memory space. Pairing get/restore allows the Vector to track access, thus knowing if norms or other operations may need to be recomputed.

Parameters

- **vec** – [**inout**] CeedVector to access
- **mem_type** – [**in**] Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy.
- **array** – [**out**] Array on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorGetArrayRead**(*CeedVector* vec, *CeedMemType* mem_type, const *CeedScalar* **array)

Get read-only access to a CeedVector via the specified memory type.

Restore access with *CeedVectorRestoreArrayRead()*.

User Functions

Parameters

- **vec** – [**in**] CeedVector to access
- **mem_type** – [**in**] Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy (possibly cached).
- **array** – [**out**] Array on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorGetArrayWrite**(*CeedVector* vec, *CeedMemType* mem_type, *CeedScalar* **array)

Get write access to a CeedVector via the specified memory type.

Restore access with *CeedVectorRestoreArray()*. All old values should be assumed to be invalid.

User Functions

Parameters

- **vec** – [**inout**] CeedVector to access
- **mem_type** – [**in**] Memory type on which to access the array.
- **array** – [**out**] Array on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorRestoreArray**(*CeedVector* vec, *CeedScalar* **array)

Restore an array obtained using *CeedVectorGetArray()* or *CeedVectorGetArrayWrite()*

User Functions

Parameters

- **vec** – [**inout**] CeedVector to restore
- **array** – [**inout**] Array of vector data

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorRestoreArrayRead**(*CeedVector* vec, const *CeedScalar* **array)

Restore an array obtained using *CeedVectorGetArrayRead()*

User Functions

Parameters

- **vec** – [**in**] CeedVector to restore
- **array** – [**inout**] Array of vector data

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorNorm**(*CeedVector* vec, *CeedNormType* norm_type, *CeedScalar* *norm)

Get the norm of a CeedVector.

Note: This operation is local to the CeedVector. This function will likely not provide the desired results for the norm of the libCEED portion of a parallel vector or a CeedVector with duplicated or hanging nodes.

User Functions

Parameters

- **vec** – [**in**] CeedVector to retrieve maximum value
- **norm_type** – [**in**] Norm type *CEED_NORM_1*, *CEED_NORM_2*, or *CEED_NORM_MAX*
- **norm** – [**out**] Variable to store norm value

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorScale**(*CeedVector* x, *CeedScalar* alpha)

Compute x = alpha x.

User Functions

Parameters

- **x** – [**inout**] vector for scaling
- **alpha** – [**in**] scaling factor

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorAXPY`**(*CeedVector* y, *CeedScalar* alpha, *CeedVector* x)

Compute y = alpha x + y.

User Functions

Parameters

- **`y`** – [**inout**] target vector for sum
- **`alpha`** – [**in**] scaling factor
- **`x`** – [**in**] second vector, must be different than y

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorAXPBY`**(*CeedVector* y, *CeedScalar* alpha, *CeedScalar* beta, *CeedVector* x)

Compute y = alpha x + beta y.

User Functions

Parameters

- **`y`** – [**inout**] target vector for sum
- **`alpha`** – [**in**] first scaling factor
- **`beta`** – [**in**] second scaling factor
- **`x`** – [**in**] second vector, must be different than y

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorPointwiseMult`**(*CeedVector* w, *CeedVector* x, *CeedVector* y)

Compute the pointwise multiplication w = x .

- y.

Any subset of x, y, and w may be the same vector.

User Functions

Parameters

- **`w`** – [**out**] target vector for the product
- **`x`** – [**in**] first vector for product
- **`y`** – [**in**] second vector for the product

Returns

An error code: 0 - success, otherwise - failure

int **`CeedVectorReciprocal`**(*CeedVector* vec)

Take the reciprocal of a CeedVector.

User Functions

Parameters

- **`vec`** – [**inout**] CeedVector to take reciprocal

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorViewRange**(*CeedVector* vec, CeedSize start, CeedSize stop, *CeedInt* step, const char *fp_fmt, FILE *stream)

View a CeedVector.

Note: It is safe to use any unsigned values for `start` or `stop` and any nonzero integer for `step`. Any portion of the provided range that is outside the range of valid indices for the CeedVector will be ignored.

User Functions

Parameters

- **`vec`** – [**in**] CeedVector to view
- **`start`** – [**in**] Index of first CeedVector entry to view
- **`stop`** – [**in**] Index of last CeedVector entry to view
- **`step`** – [**in**] Step between CeedVector entries to view
- **`fp_fmt`** – [**in**] Printing format
- **`stream`** – [**in**] Filestream to write to

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorView**(*CeedVector* vec, const char *fp_fmt, FILE *stream)

View a CeedVector.

User Functions

Parameters

- **`vec`** – [**in**] CeedVector to view
- **`fp_fmt`** – [**in**] Printing format
- **`stream`** – [**in**] Filestream to write to

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorGetCeed**(*CeedVector* vec, *Ceed* *ceed)

Get the Ceed associated with a CeedVector.

Advanced Functions

Parameters

- **`vec`** – [**in**] CeedVector to retrieve state
- **`ceed`** – [**out**] Variable to store ceed

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorGetLength**(*CeedVector* vec, CeedSize *length)

Get the length of a CeedVector.

User Functions

Parameters

- **`vec`** – [**in**] CeedVector to retrieve length
- **`length`** – [**out**] Variable to store length

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorDestroy**(*CeedVector* *vec)

Destroy a CeedVector.

User Functions

Parameters

- **vec** – [**inout**] CeedVector to destroy

Returns

An error code: 0 - success, otherwise - failure

Typedefs and Enumerations

typedef int32_t **CeedInt**

Integer type, used for indexing.

typedef float **CeedScalar**

enum **CeedCopyMode**

Conveys ownership status of arrays passed to Ceed interfaces.

Values:

enumerator **CEED_COPY_VALUES**

Implementation will copy the values and not store the passed pointer.

enumerator **CEED_USE_POINTER**

Implementation can use and modify the data provided by the user, but does not take ownership.

enumerator **CEED_OWN_POINTER**

Implementation takes ownership of the pointer and will free using *CeedFree()* when done using it.

The user should not assume that the pointer remains valid after ownership has been transferred. Note that arrays allocated using C++ operator new or other allocators cannot generally be freed using *CeedFree()*. *CeedFree()* is capable of freeing any memory that can be freed using free().

enum **CeedNormType**

Denotes type of vector norm to be computed.

Values:

enumerator **CEED_NORM_1**

$\|x\|_1 = \sum_i |x_i|$

enumerator **CEED_NORM_2**

$\|x\|_2 = \sqrt{\sum_i x_i^2}$

enumerator **CEED_NORM_MAX**

$\|x\|_\infty = \max_i |x_i|$

6.1.3 CeedElemRestriction

A *CeedElemRestriction* decomposes elements and groups the degrees of freedom (DoFs) according to the different elements they belong to.

6.1.3.1 Expressing element decomposition and degrees of freedom over a mesh

typedef struct CeedElemRestriction_private ***CeedElemRestriction**

Handle for object describing restriction to elements.

const *CeedInt* **CEED_STRIDES_BACKEND**[3] = {0}

Indicate that the stride is determined by the backend.

const *CeedElemRestriction* **CEED_ELEMRESTRICTION_NONE** = &ceed_elemrestriction_none

Argument for CeedOperatorSetField indicating that the field does not requre a CeedElemRestriction.

int **CeedElemRestrictionCreate**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* num_comp, *CeedInt* comp_stride, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, *CeedElemRestriction* *rstr)

Create a CeedElemRestriction.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedElemRestriction will be created
- **num_elem** – [**in**] Number of elements described in the *offsets* array
- **elem_size** – [**in**] Size (number of "nodes") per element
- **num_comp** – [**in**] Number of field components per interpolation node (1 for scalar fields)
- **comp_stride** – [**in**] Stride between components for the same L-vector "node". Data for node i, component j, element k can be found in the L-vector at index offsets[i + k*elem_size] + j*comp_stride.
- **l_size** – [**in**] The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – [**in**] Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – [**in**] Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – [**in**] Array of shape [*num_elem*, *elem_size*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *num_elem*. All offsets must be in the range [0, *l_size* - 1].
- **rstr** – [**out**] Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateOriented**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* num_comp, *CeedInt* comp_stride, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, const bool *orients, *CeedElemRestriction* *rstr)

Create a CeedElemRestriction with orientation signs.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedElemRestriction will be created
- **num_elem** – [**in**] Number of elements described in the *offsets* array
- **elem_size** – [**in**] Size (number of "nodes") per element
- **num_comp** – [**in**] Number of field components per interpolation node (1 for scalar fields)
- **comp_stride** – [**in**] Stride between components for the same L-vector "node". Data for node i, component j, element k can be found in the L-vector at index offsets[i + k*elem_size] + j*comp_stride.
- **l_size** – [**in**] The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – [**in**] Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – [**in**] Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – [**in**] Array of shape [*num_elem*, *elem_size*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *num_elem*. All offsets must be in the range [0, *l_size* - 1].
- **orients** – [**in**] Array of shape [*num_elem*, *elem_size*] with bool false for positively oriented and true to flip the orientation.
- **rstr** – [**out**] Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateCurlOriented**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* num_comp, *CeedInt* comp_stride, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, const CeedInt8 *curl_orients, *CeedElemRestriction* *rstr)

Create a CeedElemRestriction with a general tridiagonal transformation matrix for curl-conforming elements.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedElemRestriction will be created
- **num_elem** – [**in**] Number of elements described in the *offsets* array
- **elem_size** – [**in**] Size (number of "nodes") per element

- **num_comp** – **[in]** Number of field components per interpolation node (1 for scalar fields)
- **comp_stride** – **[in]** Stride between components for the same L-vector "node". Data for node i, component j, element k can be found in the L-vector at index offsets[i + k*elem_size] + j*comp_stride.
- **l_size** – **[in]** The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – **[in]** Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – **[in]** Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – **[in]** Array of shape [*num_elem*, *elem_size*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *num_elem*. All offsets must be in the range [0, *l_size* - 1].
- **curl_orients** – **[in]** Array of shape [*num_elem*, *3* * elem_size] representing a row-major tridiagonal matrix (curl_orients[i * 3 * elem_size] = curl_orients[(i + 1) * 3 * elem_size - 1] = 0, where 0 <= i < *num_elem*) which is applied to the element unknowns upon restriction. This orientation matrix allows for pairs of face degrees of freedom on elements for H(curl) spaces to be coupled in the element restriction operation, which is a way to resolve face orientation issues for 3D meshes (https://dl.acm.org/doi/pdf/10.1145/3524456).
- **rstr** – **[out]** Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateStrided**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* num_comp, CeedSize l_size, const *CeedInt* strides[3], *CeedElemRestriction* *rstr)

Create a strided CeedElemRestriction.

User Functions

Parameters

- **ceed** – **[in]** Ceed object where the CeedElemRestriction will be created
- **num_elem** – **[in]** Number of elements described by the restriction
- **elem_size** – **[in]** Size (number of "nodes") per element
- **num_comp** – **[in]** Number of field components per interpolation "node" (1 for scalar fields)
- **l_size** – **[in]** The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **strides** – **[in]** Array for strides between [nodes, components, elements]. Data for node i, component j, element k can be found in the L-vector at index i*strides[0] + j*strides[1] + k*strides[2]. *CEED_STRIDES_BACKEND* may be used with vectors created by a Ceed backend.
- **rstr** – **[out]** Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateAtPoints**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* num_points, *CeedInt* num_comp, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, *CeedElemRestriction* *rstr)

Create a points CeedElemRestriction, for restricting for restricting from a all local points to the current element in which they are located.

The offsets array is arranged as

element_0_start_index element_1_start_index … element_n_start_index element_n_stop_index element_0_point_0 element_0_point_1 …

Backend Developer Functions

Parameters

- **ceed** – [in] Ceed object where the CeedElemRestriction will be created
- **num_elem** – [in] Number of elements described in the *offsets* array
- **num_points** – [in] Number of points described in the *offsets* array
- **num_comp** – [in] Number of field components per interpolation node (1 for scalar fields). Components are assumed to be contiguous by point.
- **l_size** – [in] The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – [in] Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – [in] Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – [in] Array of size num_elem + 1 + num_points. The first portion of the offsets array holds the ranges of indices corresponding to each element. The second portion holds the indices for each element.
- **rstr** – [out] Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateBlocked**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* block_size, *CeedInt* num_comp, *CeedInt* comp_stride, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, *CeedElemRestriction* *rstr)

Create a blocked CeedElemRestriction, typically only called by backends.

Backend Developer Functions

Parameters

- **ceed** – [in] Ceed object where the CeedElemRestriction will be created
- **num_elem** – [in] Number of elements described in the *offsets* array
- **elem_size** – [in] Size (number of unknowns) per element
- **block_size** – [in] Number of elements in a block
- **num_comp** – [in] Number of field components per interpolation node (1 for scalar fields)

- **comp_stride** – [in] Stride between components for the same L-vector "node". Data for node i, component j, element k can be found in the L-vector at index offsets[i + k*elem_size] + j*comp_stride.
- **l_size** – [in] The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – [in] Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – [in] Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – [in] Array of shape [*num_elem*, *elem_size*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *num_elem*. All offsets must be in the range [0, *l_size* - 1]. The backend will permute and pad this array to the desired ordering for the blocksize, which is typically given by the backend. The default reordering is to interlace elements.
- **rstr** – [out] Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateBlockedOriented**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* block_size, *CeedInt* num_comp, *CeedInt* comp_stride, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, const bool *orients, *CeedElemRestriction* *rstr)

Create a blocked oriented CeedElemRestriction, typically only called by backends.

Backend Developer Functions

Parameters

- **ceed** – [in] Ceed object where the CeedElemRestriction will be created.
- **num_elem** – [in] Number of elements described in the *offsets* array.
- **elem_size** – [in] Size (number of unknowns) per element
- **block_size** – [in] Number of elements in a block
- **num_comp** – [in] Number of field components per interpolation node (1 for scalar fields)
- **comp_stride** – [in] Stride between components for the same L-vector "node". Data for node i, component j, element k can be found in the L-vector at index offsets[i + k*elem_size] + j*comp_stride.
- **l_size** – [in] The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – [in] Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – [in] Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – [in] Array of shape [*num_elem*, *elem_size*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *num_elem*. All offsets must be in the range [0, *l_size* - 1]. The backend will permute and pad this array to the desired ordering for the blocksize, which is typically given by the backend. The default reordering is to interlace elements.

- **orients** – [in] Array of shape [*num_elem*, *elem_size*] with bool false for positively oriented and true to flip the orientation. Will also be permuted and padded similarly to *offsets*.
- **rstr** – [out] Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateBlockedCurlOriented**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* block_size, *CeedInt* num_comp, *CeedInt* comp_stride, CeedSize l_size, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, const *CeedInt* *offsets, const CeedInt8 *curl_orients, *CeedElemRestriction* *rstr)

Create a blocked curl-oriented CeedElemRestriction, typically only called by backends.

Backend Developer Functions

Parameters

- **ceed** – [in] Ceed object where the CeedElemRestriction will be created.
- **num_elem** – [in] Number of elements described in the *offsets* array.
- **elem_size** – [in] Size (number of unknowns) per element
- **block_size** – [in] Number of elements in a block
- **num_comp** – [in] Number of field components per interpolation node (1 for scalar fields)
- **comp_stride** – [in] Stride between components for the same L-vector "node". Data for node i, component j, element k can be found in the L-vector at index offsets[i + k*elem_size] + j*comp_stride.
- **l_size** – [in] The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **mem_type** – [in] Memory type of the *offsets* array, see CeedMemType
- **copy_mode** – [in] Copy mode for the *offsets* array, see CeedCopyMode
- **offsets** – [in] Array of shape [*num_elem*, *elem_size*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *num_elem*. All offsets must be in the range [0, *l_size* - 1]. The backend will permute and pad this array to the desired ordering for the blocksize, which is typically given by the backend. The default reordering is to interlace elements.
- **curl_orients** – [in] Array of shape [*num_elem*, *3* * elem_size] representing a row-major tridiagonal matrix (curl_orients[i * 3 * elem_size] = curl_orients[(i + 1) * 3 * elem_size - 1] = 0, where 0 <= i < *num_elem*) which is applied to the element unknowns upon restriction. This orientation matrix allows for pairs of face degrees of freedom on elements for H(curl) spaces to be coupled in the element restriction operation, which is a way to resolve face orientation issues for 3D meshes (https://dl.acm.org/doi/pdf/10.1145/3524456). Will also be permuted and padded similarly to *offsets*.
- **rstr** – [out] Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateBlockedStrided**(*Ceed* ceed, *CeedInt* num_elem, *CeedInt* elem_size, *CeedInt* block_size, *CeedInt* num_comp, CeedSize l_size, const *CeedInt* strides[3], *CeedElemRestriction* *rstr)

Create a blocked strided CeedElemRestriction, typically only called by backends.

User Functions

Parameters

- **ceed** – **[in]** Ceed object where the CeedElemRestriction will be created
- **num_elem** – **[in]** Number of elements described by the restriction
- **elem_size** – **[in]** Size (number of "nodes") per element
- **block_size** – **[in]** Number of elements in a block
- **num_comp** – **[in]** Number of field components per interpolation node (1 for scalar fields)
- **l_size** – **[in]** The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- **strides** – **[in]** Array for strides between [nodes, components, elements]. Data for node i, component j, element k can be found in the L-vector at index i*strides[0] + j*strides[1] + k*strides[2]. *CEED_STRIDES_BACKEND* may be used with vectors created by a Ceed backend.
- **rstr** – **[out]** Address of the variable where the newly created CeedElemRestriction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateUnsignedCopy**(*CeedElemRestriction* rstr, *CeedElemRestriction* *rstr_unsigned)

Copy the pointer to a CeedElemRestriction and set *CeedElemRestrictionApply()* implementation to use the unsigned version.

Both pointers should be destroyed with *CeedElemRestrictionDestroy()*.

User Functions

Parameters

- **rstr** – **[in]** CeedElemRestriction to create unsigned reference to
- **rstr_unsigned** – **[inout]** Variable to store unsigned CeedElemRestriction

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionCreateUnorientedCopy**(*CeedElemRestriction* rstr, *CeedElemRestriction* *rstr_unoriented)

Copy the pointer to a CeedElemRestriction and set *CeedElemRestrictionApply()* implementation to use the unoriented version.

Both pointers should be destroyed with *CeedElemRestrictionDestroy()*.

User Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction to create unoriented reference to
- **`rstr_unoriented`** – [**inout**] Variable to store unoriented CeedElemRestriction

Returns

An error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionReferenceCopy`**(*CeedElemRestriction* rstr, *CeedElemRestriction* *rstr_copy)

Copy the pointer to a CeedElemRestriction.

Both pointers should be destroyed with *`CeedElemRestrictionDestroy()`*.

Note: If the value of `rstr_copy` passed into this function is non-NULL, then it is assumed that `rstr_copy` is a pointer to a CeedElemRestriction. This CeedElemRestriction will be destroyed if `rstr_copy` is the only reference to this CeedElemRestriction.

User Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction to copy reference to
- **`rstr_copy`** – [**inout**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionCreateVector`**(*CeedElemRestriction* rstr, *CeedVector* *l_vec, *CeedVector* *e_vec)

Create CeedVectors associated with a CeedElemRestriction.

User Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction
- **`l_vec`** – [**out**] The address of the L-vector to be created, or NULL
- **`e_vec`** – [**out**] The address of the E-vector to be created, or NULL

Returns

An error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionApply`**(*CeedElemRestriction* rstr, *CeedTransposeMode* t_mode, *CeedVector* u, *CeedVector* ru, *CeedRequest* *request)

Restrict an L-vector to an E-vector or apply its transpose.

User Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction
- **`t_mode`** – [**in**] Apply restriction or transpose
- **`u`** – [**in**] Input vector (of size *l_size* when t_mode=*CEED_NOTRANSPOSE*)
- **`ru`** – [**out**] Output vector (of shape [*num_elem* * *elem_size*] when t_mode=*CEED_NOTRANSPOSE*). Ordering of the e-vector is decided by the backend.
- **`request`** – [**in**] Request or CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionApplyAtPointsInElement**(*CeedElemRestriction* rstr, *CeedInt* elem, *CeedTransposeMode* t_mode, *CeedVector* u, *CeedVector* ru, *CeedRequest* *request)

Restrict an L-vector of points to a single element or apply its transpose.

User Functions

Parameters

- **rstr** – **[in]** CeedElemRestriction
- **elem** – **[in]** Element number in range 0..*num_elem*
- **t_mode** – **[in]** Apply restriction or transpose
- **u** – **[in]** Input vector (of size *l_size* when t_mode=*CEED_NOTRANSPOSE*)
- **ru** – **[out]** Output vector (of shape [*num_elem* * *elem_size*] when t_mode=*CEED_NOTRANSPOSE*). Ordering of the e-vector is decided by the backend.
- **request** – **[in]** Request or CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionApplyBlock**(*CeedElemRestriction* rstr, *CeedInt* block, *CeedTransposeMode* t_mode, *CeedVector* u, *CeedVector* ru, *CeedRequest* *request)

Restrict an L-vector to a block of an E-vector or apply its transpose.

Backend Developer Functions

Parameters

- **rstr** – **[in]** CeedElemRestriction
- **block** – **[in]** Block number to restrict to/from, i.e. block=0 will handle elements [0 : block_size] and block=3 will handle elements [3*block_size : 4*block_size]
- **t_mode** – **[in]** Apply restriction or transpose
- **u** – **[in]** Input vector (of size *l_size* when t_mode=*CEED_NOTRANSPOSE*)
- **ru** – **[out]** Output vector (of shape [*block_size* * *elem_size*] when t_mode=*CEED_NOTRANSPOSE*). Ordering of the e-vector is decided by the backend.
- **request** – **[in]** Request or CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetCeed**(*CeedElemRestriction* rstr, *Ceed* *ceed)

Get the Ceed associated with a CeedElemRestriction.

Advanced Functions

Parameters

- **rstr** – **[in]** CeedElemRestriction
- **ceed** – **[out]** Variable to store Ceed

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetCompStride**(*CeedElemRestriction* rstr, *CeedInt* *comp_stride)

Get the L-vector component stride.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **comp_stride** – [**out**] Variable to store component stride

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetNumElements**(*CeedElemRestriction* rstr, *CeedInt* *num_elem)

Get the total number of elements in the range of a CeedElemRestriction.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **num_elem** – [**out**] Variable to store number of elements

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetElementSize**(*CeedElemRestriction* rstr, *CeedInt* *elem_size)

Get the size of elements in the CeedElemRestriction.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **elem_size** – [**out**] Variable to store size of elements

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetNumPoints**(*CeedElemRestriction* rstr, *CeedInt* *num_points)

Get the number of points in the l-vector for a points CeedElemRestriction.

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **num_points** – [**out**] The number of points in the l-vector

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetNumPointsInElement**(*CeedElemRestriction* rstr, *CeedInt* elem, *CeedInt* *num_points)

Get the number of points in an element of a points CeedElemRestriction.

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **elem** – [**in**] Index number of element to retrieve the number of points for
- **num_points** – [**out**] The number of points in the element at index elem

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetMaxPointsInElement**(*CeedElemRestriction* rstr, *CeedInt* *max_points)

Get the maximum number of points in an element for a CeedElemRestriction at points.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **max_points** – [**out**] Variable to store size of elements

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetLVectorSize**(*CeedElemRestriction* rstr, CeedSize *l_size)

Get the size of the l-vector for a CeedElemRestriction.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **l_size** – [**out**] Variable to store number of nodes

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetNumComponents**(*CeedElemRestriction* rstr, *CeedInt* *num_comp)

Get the number of components in the elements of a CeedElemRestriction.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **num_comp** – [**out**] Variable to store number of components

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetNumBlocks**(*CeedElemRestriction* rstr, *CeedInt* *num_block)

Get the number of blocks in a CeedElemRestriction.

Advanced Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **num_block** – [**out**] Variable to store number of blocks

Returns

An error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionGetBlockSize`**(*CeedElemRestriction* rstr, *CeedInt* *block_size)

Get the size of blocks in the CeedElemRestriction.

Advanced Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction
- **`block_size`** – [**out**] Variable to store size of blocks

Returns

An error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionGetMultiplicity`**(*CeedElemRestriction* rstr, *CeedVector* mult)

Get the multiplicity of nodes in a CeedElemRestriction.

User Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction
- **`mult`** – [**out**] Vector to store multiplicity (of size l_size)

Returns

An error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionView`**(*CeedElemRestriction* rstr, FILE *stream)

View a CeedElemRestriction.

User Functions

Parameters

- **`rstr`** – [**in**] CeedElemRestriction to view
- **`stream`** – [**in**] Stream to write; typically stdout/stderr or a file

Returns

Error code: 0 - success, otherwise - failure

int **`CeedElemRestrictionDestroy`**(*CeedElemRestriction* *rstr)

Destroy a CeedElemRestriction.

User Functions

Parameters

- **`rstr`** – [**inout**] CeedElemRestriction to destroy

Returns

An error code: 0 - success, otherwise - failure

6.1.4 CeedBasis

A *CeedBasis* defines the discrete finite element basis and associated quadrature rule.

6.1.4.1 Discrete element bases and quadrature

typedef struct CeedBasis_private ***CeedBasis**

Handle for object describing discrete finite element evaluations.

const *CeedBasis* **CEED_BASIS_NONE** = &ceed_basis_none

Argument for CeedOperatorSetField indicating that the field does not require a CeedBasis.

const *CeedBasis* **CEED_BASIS_COLLOCATED** = &ceed_basis_none

This feature will be removed. Use CEED_BASIS_NONE.

int **CeedBasisCreateTensorH1**(*Ceed* ceed, *CeedInt* dim, *CeedInt* num_comp, *CeedInt* P_1d, *CeedInt* Q_1d, const *CeedScalar* *interp_1d, const *CeedScalar* *grad_1d, const *CeedScalar* *q_ref_1d, const *CeedScalar* *q_weight_1d, *CeedBasis* *basis)

Create a tensor-product basis for H^1 discretizations.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedBasis will be created
- **dim** – [**in**] Topological dimension
- **num_comp** – [**in**] Number of field components (1 for scalar fields)
- **P_1d** – [**in**] Number of nodes in one dimension
- **Q_1d** – [**in**] Number of quadrature points in one dimension
- **interp_1d** – [**in**] Row-major (Q_1d * P_1d) matrix expressing the values of nodal basis functions at quadrature points
- **grad_1d** – [**in**] Row-major (Q_1d * P_1d) matrix expressing derivatives of nodal basis functions at quadrature points
- **q_ref_1d** – [**in**] Array of length Q_1d holding the locations of quadrature points on the 1D reference element [-1, 1]
- **q_weight_1d** – [**in**] Array of length Q_1d holding the quadrature weights on the reference element
- **basis** – [**out**] Address of the variable where the newly created CeedBasis will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisCreateTensorH1Lagrange**(*Ceed* ceed, *CeedInt* dim, *CeedInt* num_comp, *CeedInt* P, *CeedInt* Q, *CeedQuadMode* quad_mode, *CeedBasis* *basis)

Create a tensor-product Lagrange basis.

User Functions

Parameters

- **ceed** – [in] Ceed object where the CeedBasis will be created
- **dim** – [in] Topological dimension of element
- **num_comp** – [in] Number of field components (1 for scalar fields)
- **P** – [in] Number of Gauss-Lobatto nodes in one dimension. The polynomial degree of the resulting Q_k element is k=P-1.
- **Q** – [in] Number of quadrature points in one dimension.
- **quad_mode** – [in] Distribution of the Q quadrature points (affects order of accuracy for the quadrature)
- **basis** – [out] Address of the variable where the newly created CeedBasis will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisCreateH1**(*Ceed* ceed, *CeedElemTopology* topo, *CeedInt* num_comp, *CeedInt* num_nodes, *CeedInt* num_qpts, const *CeedScalar* *interp, const *CeedScalar* *grad, const *CeedScalar* *q_ref, const *CeedScalar* *q_weight, *CeedBasis* *basis)

Create a non tensor-product basis for H^1 discretizations.

User Functions

Parameters

- **ceed** – [in] Ceed object where the CeedBasis will be created
- **topo** – [in] Topology of element, e.g. hypercube, simplex, ect
- **num_comp** – [in] Number of field components (1 for scalar fields)
- **num_nodes** – [in] Total number of nodes
- **num_qpts** – [in] Total number of quadrature points
- **interp** – [in] Row-major (num_qpts * num_nodes) matrix expressing the values of nodal basis functions at quadrature points
- **grad** – [in] Row-major (dim * num_qpts * num_nodes) matrix expressing derivatives of nodal basis functions at quadrature points
- **q_ref** – [in] Array of length num_qpts * dim holding the locations of quadrature points on the reference element
- **q_weight** – [in] Array of length num_qpts holding the quadrature weights on the reference element
- **basis** – [out] Address of the variable where the newly created CeedBasis will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisCreateHdiv**(*Ceed* ceed, *CeedElemTopology* topo, *CeedInt* num_comp, *CeedInt* num_nodes, *CeedInt* num_qpts, const *CeedScalar* *interp, const *CeedScalar* *div, const *CeedScalar* *q_ref, const *CeedScalar* *q_weight, *CeedBasis* *basis)

Create a non tensor-product basis for $H(\mathrm{div})$ discretizations.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedBasis will be created
- **topo** – [**in**] Topology of element (`CEED_TOPOLOGY_QUAD`, `CEED_TOPOLOGY_PRISM`, etc.), dimension of which is used in some array sizes below
- **num_comp** – [**in**] Number of components (usually 1 for vectors in H(div) bases)
- **num_nodes** – [**in**] Total number of nodes (dofs per element)
- **num_qpts** – [**in**] Total number of quadrature points
- **interp** – [**in**] Row-major (dim * num_qpts * num_nodes) matrix expressing the values of basis functions at quadrature points
- **div** – [**in**] Row-major (num_qpts * num_nodes) matrix expressing divergence of basis functions at quadrature points
- **q_ref** – [**in**] Array of length num_qpts * dim holding the locations of quadrature points on the reference element
- **q_weight** – [**in**] Array of length num_qpts holding the quadrature weights on the reference element
- **basis** – [**out**] Address of the variable where the newly created CeedBasis will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisCreateHcurl**(*Ceed* ceed, *CeedElemTopology* topo, *CeedInt* num_comp, *CeedInt* num_nodes, *CeedInt* num_qpts, const *CeedScalar* *interp, const *CeedScalar* *curl, const *CeedScalar* *q_ref, const *CeedScalar* *q_weight, *CeedBasis* *basis)

Create a non tensor-product basis for $H(\mathrm{curl})$ discretizations.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedBasis will be created
- **topo** – [**in**] Topology of element (`CEED_TOPOLOGY_QUAD`, `CEED_TOPOLOGY_PRISM`, etc.), dimension of which is used in some array sizes below
- **num_comp** – [**in**] Number of components (usually 1 for vectors in H(curl) bases)
- **num_nodes** – [**in**] Total number of nodes (dofs per element)
- **num_qpts** – [**in**] Total number of quadrature points
- **interp** – [**in**] Row-major (dim * num_qpts * num_nodes) matrix expressing the values of basis functions at quadrature points
- **curl** – [**in**] Row-major (curl_comp * num_qpts * num_nodes, curl_comp = 1 if dim < 3 else dim) matrix expressing curl of basis functions at quadrature points
- **q_ref** – [**in**] Array of length num_qpts * dim holding the locations of quadrature points on the reference element
- **q_weight** – [**in**] Array of length num_qpts holding the quadrature weights on the reference element
- **basis** – [**out**] Address of the variable where the newly created CeedBasis will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisCreateProjection**(*CeedBasis* basis_from, *CeedBasis* basis_to, *CeedBasis* *basis_project)

Create a CeedBasis for projection from the nodes of `basis_from` to the nodes of `basis_to`.

Only `CEED_EVAL_INTERP` will be valid for the new basis, `basis_project`. For H^1 spaces, `CEED_EVAL_GRAD` will also be valid. The interpolation is given by `interp_project = interp_to^+ * interp_from`, where the pseudoinverse `interp_to^+` is given by QR factorization. The gradient (for the H^1 case) is given by `grad_project = interp_to^+ * grad_from`.

Note: `basis_from` and `basis_to` must have compatible quadrature spaces.

Note: `basis_project` will have the same number of components as `basis_from`, regardless of the number of components that `basis_to` has. If `basis_from` has 3 components and `basis_to` has 5 components, then `basis_project` will have 3 components.

User Functions

Parameters

- **`basis_from`** – [**in**] CeedBasis to prolong from
- **`basis_to`** – [**in**] CeedBasis to prolong to
- **`basis_project`** – [**out**] Address of the variable where the newly created CeedBasis will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisReferenceCopy**(*CeedBasis* basis, *CeedBasis* *basis_copy)

Copy the pointer to a CeedBasis.

Note: If the value of `basis_copy` passed into this function is non-NULL, then it is assumed that `basis_copy` is a pointer to a CeedBasis. This CeedBasis will be destroyed if `basis_copy` is the only reference to this CeedBasis.

User Functions

Parameters

- **`basis`** – [**in**] CeedBasis to copy reference to
- **`basis_copy`** – [**inout**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisView**(*CeedBasis* basis, FILE *stream)

View a CeedBasis.

User Functions

Parameters

- **`basis`** – [**in**] CeedBasis to view
- **`stream`** – [**in**] Stream to view to, e.g., stdout

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisApply**(*CeedBasis* basis, *CeedInt* num_elem, *CeedTransposeMode* t_mode, *CeedEvalMode* eval_mode, *CeedVector* u, *CeedVector* v)

Apply basis evaluation from nodes to quadrature points or vice versa.

User Functions

Parameters

- **basis** – [**in**] CeedBasis to evaluate
- **num_elem** – [**in**] The number of elements to apply the basis evaluation to; the backend will specify the ordering in *CeedElemRestrictionCreateBlocked()*
- **t_mode** – [**in**] *CEED_NOTRANSPOSE* to evaluate from nodes to quadrature points; *CEED_TRANSPOSE* to apply the transpose, mapping from quadrature points to nodes
- **eval_mode** – [**in**] *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_DIV* to use divergence, *CEED_EVAL_CURL* to use curl, *CEED_EVAL_WEIGHT* to use quadrature weights.
- **u** – [**in**] Input CeedVector
- **v** – [**out**] Output CeedVector

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisApplyAtPoints**(*CeedBasis* basis, *CeedInt* num_points, *CeedTransposeMode* t_mode, *CeedEvalMode* eval_mode, *CeedVector* x_ref, *CeedVector* u, *CeedVector* v)

Apply basis evaluation from nodes to arbitrary points.

User Functions

Parameters

- **basis** – [**in**] CeedBasis to evaluate
- **num_points** – [**in**] The number of points to apply the basis evaluation to
- **t_mode** – [**in**] *CEED_NOTRANSPOSE* to evaluate from nodes to points; *CEED_TRANSPOSE* to apply the transpose, mapping from points to nodes
- **eval_mode** – [**in**] *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients
- **x_ref** – [**in**] CeedVector holding reference coordinates of each point
- **u** – [**in**] Input CeedVector, of length `num_nodes * num_comp` for `CEED_NOTRANSPOSE`
- **v** – [**out**] Output CeedVector, of length `num_points * num_q_comp` for `CEED_NOTRANSPOSE` with `CEED_EVAL_INTERP`

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetCeed**(*CeedBasis* basis, *Ceed* *ceed)

Get Ceed associated with a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **ceed** – [**out**] Variable to store Ceed

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetDimension**(*CeedBasis* basis, *CeedInt* *dim)

Get dimension for given CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **dim** – [**out**] Variable to store dimension of basis

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetTopology**(*CeedBasis* basis, *CeedElemTopology* *topo)

Get topology for given CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **topo** – [**out**] Variable to store topology of basis

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetNumComponents**(*CeedBasis* basis, *CeedInt* *num_comp)

Get number of components for given CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **num_comp** – [**out**] Variable to store number of components of basis

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetNumNodes**(*CeedBasis* basis, *CeedInt* *P)

Get total number of nodes (in dim dimensions) of a CeedBasis.

Utility Functions

Parameters

- **basis** – [**in**] CeedBasis
- **P** – [**out**] Variable to store number of nodes

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetNumNodes1D**(*CeedBasis* basis, *CeedInt* *P_1d)

Get total number of nodes (in 1 dimension) of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **P_1d** – [**out**] Variable to store number of nodes

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetNumQuadraturePoints**(*CeedBasis* basis, *CeedInt* *Q)

Get total number of quadrature points (in dim dimensions) of a CeedBasis.

Utility Functions

Parameters

- **basis** – [**in**] CeedBasis
- **Q** – [**out**] Variable to store number of quadrature points

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetNumQuadraturePoints1D**(*CeedBasis* basis, *CeedInt* *Q_1d)

Get total number of quadrature points (in 1 dimension) of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **Q_1d** – [**out**] Variable to store number of quadrature points

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetQRef**(*CeedBasis* basis, const *CeedScalar* **q_ref)

Get reference coordinates of quadrature points (in dim dimensions) of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **q_ref** – [**out**] Variable to store reference coordinates of quadrature points

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetQWeights**(*CeedBasis* basis, const *CeedScalar* **q_weight)

Get quadrature weights of quadrature points (in dim dimensions) of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **q_weight** – [**out**] Variable to store quadrature weights

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetInterp**(*CeedBasis* basis, const *CeedScalar* **interp)

Get interpolation matrix of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis

- **interp** – [**out**] Variable to store interpolation matrix

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetInterp1D**(*CeedBasis* basis, const *CeedScalar* **interp_1d)

Get 1D interpolation matrix of a tensor product CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **interp_1d** – [**out**] Variable to store interpolation matrix

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetGrad**(*CeedBasis* basis, const *CeedScalar* **grad)

Get gradient matrix of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **grad** – [**out**] Variable to store gradient matrix

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetGrad1D**(*CeedBasis* basis, const *CeedScalar* **grad_1d)

Get 1D gradient matrix of a tensor product CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **grad_1d** – [**out**] Variable to store gradient matrix

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetDiv**(*CeedBasis* basis, const *CeedScalar* **div)

Get divergence matrix of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **div** – [**out**] Variable to store divergence matrix

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetCurl**(*CeedBasis* basis, const *CeedScalar* **curl)

Get curl matrix of a CeedBasis.

Advanced Functions

Parameters

- **basis** – [**in**] CeedBasis
- **curl** – [**out**] Variable to store curl matrix

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisDestroy**(*CeedBasis* *basis)

Destroy a CeedBasis.

User Functions

Parameters

- **basis** – [**inout**] CeedBasis to destroy

Returns
An error code: 0 - success, otherwise - failure

int **CeedGaussQuadrature**(*CeedInt* Q, *CeedScalar* *q_ref_1d, *CeedScalar* *q_weight_1d)

Construct a Gauss-Legendre quadrature.

Utility Functions

Parameters

- **Q** – [**in**] Number of quadrature points (integrates polynomials of degree 2*Q-1 exactly)
- **q_ref_1d** – [**out**] Array of length Q to hold the abscissa on [-1, 1]
- **q_weight_1d** – [**out**] Array of length Q to hold the weights

Returns
An error code: 0 - success, otherwise - failure

int **CeedLobattoQuadrature**(*CeedInt* Q, *CeedScalar* *q_ref_1d, *CeedScalar* *q_weight_1d)

Construct a Gauss-Legendre-Lobatto quadrature.

Utility Functions

Parameters

- **Q** – [**in**] Number of quadrature points (integrates polynomials of degree 2*Q-3 exactly)
- **q_ref_1d** – [**out**] Array of length Q to hold the abscissa on [-1, 1]
- **q_weight_1d** – [**out**] Array of length Q to hold the weights

Returns
An error code: 0 - success, otherwise - failure

Typedefs and Enumerations

enum **CeedTransposeMode**

Denotes whether a linear transformation or its transpose should be applied.

Values:

enumerator **`CEED_NOTRANSPOSE`**

Apply the linear transformation.

enumerator **`CEED_TRANSPOSE`**

Apply the transpose.

enum **`CeedEvalMode`**

Basis evaluation mode.

Values:

enumerator **`CEED_EVAL_NONE`**

Perform no evaluation (either because there is no data or it is already at quadrature points)

enumerator **`CEED_EVAL_INTERP`**

Interpolate from nodes to quadrature points.

enumerator **`CEED_EVAL_GRAD`**

Evaluate gradients at quadrature points from input in the basis.

enumerator **`CEED_EVAL_DIV`**

Evaluate divergence at quadrature points from input in the basis.

enumerator **`CEED_EVAL_CURL`**

Evaluate curl at quadrature points from input in the basis.

enumerator **`CEED_EVAL_WEIGHT`**

Using no input, evaluate quadrature weights on the reference element.

enum **`CeedQuadMode`**

Type of quadrature; also used for location of nodes.

Values:

enumerator **`CEED_GAUSS`**

Gauss-Legendre quadrature.

enumerator **`CEED_GAUSS_LOBATTO`**

Gauss-Legendre-Lobatto quadrature.

enum **`CeedElemTopology`**

Type of basis shape to create non-tensor element basis.

Dimension can be extracted with bitwise AND (CeedElemTopology & 2**(dim + 2)) == TRUE

Values:

enumerator **CEED_TOPOLOGY_LINE**

Line.

enumerator **CEED_TOPOLOGY_TRIANGLE**

Triangle - 2D shape.

enumerator **CEED_TOPOLOGY_QUAD**

Quadralateral - 2D shape.

enumerator **CEED_TOPOLOGY_TET**

Tetrahedron - 3D shape.

enumerator **CEED_TOPOLOGY_PYRAMID**

Pyramid - 3D shape.

enumerator **CEED_TOPOLOGY_PRISM**

Prism - 3D shape.

enumerator **CEED_TOPOLOGY_HEX**

Hexehedron - 3D shape.

6.1.5 CeedQFunction

A *CeedQFunction* represents the spatial terms of the point-wise functions describing the physics at the quadrature points.

6.1.5.1 Resolution/space-independent weak forms and quadrature-based operations

typedef struct CeedQFunction_private ***CeedQFunction**

Handle for object describing functions evaluated independently at quadrature points.

typedef struct CeedQFunctionContext_private ***CeedQFunctionContext**

Handle for object describing context data for CeedQFunctions.

typedef struct CeedContextFieldLabel_private ***CeedContextFieldLabel**

Handle for object describing registered fields for CeedQFunctionContext.

const *CeedQFunction* **CEED_QFUNCTION_NONE** = &ceed_qfunction_none

int **CeedQFunctionCreateInterior**(*Ceed* ceed, *CeedInt* vec_length, CeedQFunctionUser f, const char *source, *CeedQFunction* *qf)

Create a CeedQFunction for evaluating interior (volumetric) terms.

See Public API for CeedQFunction for details on the call-back function *f's* arguments.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedQFunction will be created
- **vec_length** – [**in**] Vector length. Caller must ensure that number of quadrature points is a multiple of vec_length.
- **f** – [**in**] Function pointer to evaluate action at quadrature points. See Public API for CeedQFunction.
- **source** – [**in**] Absolute path to source of QFunction, "\abs_path\file.h:function_name". The entire source file must only contain constructs supported by all targeted backends (i.e. CUDA for `/gpu/cuda`, OpenCL/SYCL for `/gpu/sycl`, etc.). The entire contents of this file and all locally included files are used during JiT compilation for GPU backends. All source files must be at the provided filepath at runtime for JiT to function.
- **qf** – [**out**] Address of the variable where the newly created CeedQFunction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionCreateInteriorByName**(*Ceed* ceed, const char *name, *CeedQFunction* *qf)

Create a CeedQFunction for evaluating interior (volumetric) terms by name.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedQFunction will be created
- **name** – [**in**] Name of QFunction to use from gallery
- **qf** – [**out**] Address of the variable where the newly created CeedQFunction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionCreateIdentity**(*Ceed* ceed, *CeedInt* size, *CeedEvalMode* in_mode, *CeedEvalMode* out_mode, *CeedQFunction* *qf)

Create an identity CeedQFunction.

Inputs are written into outputs in the order given. This is useful for CeedOperators that can be represented with only the action of a CeedElemRestriction and CeedBasis, such as restriction and prolongation operators for p-multigrid. Backends may optimize CeedOperators with this CeedQFunction to avoid the copy of input data to output fields by using the same memory location for both.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedQFunction will be created
- **size** – [**in**] Size of the QFunction fields
- **in_mode** – [**in**] CeedEvalMode for input to CeedQFunction
- **out_mode** – [**in**] CeedEvalMode for output to CeedQFunction
- **qf** – [**out**] Address of the variable where the newly created CeedQFunction will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionReferenceCopy**(*CeedQFunction* qf, *CeedQFunction* *qf_copy)

Copy the pointer to a CeedQFunction.

Both pointers should be destroyed with `CeedQFunctionDestroy()`.

Note: If the value of `qf_copy` passed to this function is non-NULL, then it is assumed that `*qf_copy` is a pointer to a CeedQFunction. This CeedQFunction will be destroyed if `*qf_copy` is the only reference to this CeedQFunction.

User Functions

Parameters

- **qf** – [**in**] CeedQFunction to copy reference to
- **qf_copy** – [**out**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionAddInput**(*CeedQFunction* qf, const char *field_name, *CeedInt* size, *CeedEvalMode* eval_mode)

Add a CeedQFunction input.

User Functions

Parameters

- **qf** – [**inout**] CeedQFunction
- **field_name** – [**in**] Name of QFunction field
- **size** – [**in**] Size of QFunction field, (num_comp * 1) for *CEED_EVAL_NONE*, (num_comp * 1) for *CEED_EVAL_INTERP* for an H^1 space or (num_comp * dim) for an H(div) or H(curl) space, (num_comp * dim) for *CEED_EVAL_GRAD*, or (num_comp * 1) for *CEED_EVAL_DIV*, and (num_comp * curl_dim) with curl_dim = 1 if dim < 3 else dim for *CEED_EVAL_CURL*.
- **eval_mode** – [**in**] *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_DIV* to use divergence, *CEED_EVAL_CURL* to use curl.

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionAddOutput**(*CeedQFunction* qf, const char *field_name, *CeedInt* size, *CeedEvalMode* eval_mode)

Add a CeedQFunction output.

User Functions

Parameters

- **qf** – [**inout**] CeedQFunction
- **field_name** – [**in**] Name of QFunction field
- **size** – [**in**] Size of QFunction field, (num_comp * 1) for *CEED_EVAL_NONE*, (num_comp * 1) for *CEED_EVAL_INTERP* for an H^1 space or (num_comp * dim) for an H(div) or H(curl) space, (num_comp * dim) for *CEED_EVAL_GRAD*, or

(num_comp * 1) for *CEED_EVAL_DIV*, and (num_comp * curl_dim) with curl_dim = 1 if dim < 3 else dim for *CEED_EVAL_CURL*.

- **eval_mode** – [**in**] *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_DIV* to use divergence, *CEED_EVAL_CURL* to use curl.

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetFields**(*CeedQFunction* qf, *CeedInt* *num_input_fields, *CeedQFunctionField* **input_fields, *CeedInt* *num_output_fields, *CeedQFunctionField* **output_fields)

Get the CeedQFunctionFields of a CeedQFunction.

Note: Calling this function asserts that setup is complete and sets the CeedQFunction as immutable.

Advanced Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **num_input_fields** – [**out**] Variable to store number of input fields
- **input_fields** – [**out**] Variable to store input fields
- **num_output_fields** – [**out**] Variable to store number of output fields
- **output_fields** – [**out**] Variable to store output fields

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionFieldGetName**(*CeedQFunctionField* qf_field, char **field_name)

Get the name of a CeedQFunctionField.

Advanced Functions

Parameters

- **qf_field** – [**in**] CeedQFunctionField
- **field_name** – [**out**] Variable to store the field name

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionFieldGetSize**(*CeedQFunctionField* qf_field, *CeedInt* *size)

Get the number of components of a CeedQFunctionField.

Advanced Functions

Parameters

- **qf_field** – [**in**] CeedQFunctionField
- **size** – [**out**] Variable to store the size of the field

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionFieldGetEvalMode**(*CeedQFunctionField* qf_field, *CeedEvalMode* *eval_mode)

Get the CeedEvalMode of a CeedQFunctionField.

Advanced Functions

Parameters

- **`qf_field`** – [**in**] CeedQFunctionField
- **`eval_mode`** – [**out**] Variable to store the field evaluation mode

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionSetContext`**(*CeedQFunction* qf, *CeedQFunctionContext* ctx)

Set global context for a CeedQFunction.

User Functions

Parameters

- **`qf`** – [**inout**] CeedQFunction
- **`ctx`** – [**in**] Context data to set

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionSetContextWritable`**(*CeedQFunction* qf, bool is_writable)

Set writability of CeedQFunctionContext when calling the `CeedQFunctionUser`.

The default value is `is_writable == true`.

Setting `is_writable == true` indicates the `CeedQFunctionUser` writes into the CeedQFunctionContextData and requires memory syncronization after calling *`CeedQFunctionApply()`*.

Setting 'is_writable == false' asserts that `CeedQFunctionUser` does not modify the CeedQFunctionContextData. Violating this assertion may lead to inconsistent data.

Setting `is_writable == false` may offer a performance improvement on GPU backends.

User Functions

Parameters

- **`qf`** – [**inout**] CeedQFunction
- **`is_writable`** – [**in**] Writability status

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionSetUserFlopsEstimate`**(*CeedQFunction* qf, CeedSize flops)

Set estimated number of FLOPs per quadrature required to apply QFunction.

Backend Developer Functions

Parameters

- **`qf`** – [**in**] QFunction to estimate FLOPs for
- **`flops`** – [**out**] FLOPs per quadrature point estimate

int **`CeedQFunctionView`**(*CeedQFunction* qf, FILE *stream)

View a CeedQFunction.

User Functions

Parameters

- **`qf`** – [**in**] CeedQFunction to view

- **stream** – [**in**] Stream to write; typically stdout/stderr or a file

Returns
Error code: 0 - success, otherwise - failure

int **CeedQFunctionGetCeed**(*CeedQFunction* qf, *Ceed* *ceed)

Get the Ceed associated with a CeedQFunction.

Advanced Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **ceed** – [**out**] Variable to store Ceed

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionApply**(*CeedQFunction* qf, *CeedInt* Q, *CeedVector* *u, *CeedVector* *v)

Apply the action of a CeedQFunction.

Note: Calling this function asserts that setup is complete and sets the CeedQFunction as immutable.

User Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **Q** – [**in**] Number of quadrature points
- **u** – [**in**] Array of input CeedVectors
- **v** – [**out**] Array of output CeedVectors

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionDestroy**(*CeedQFunction* *qf)

Destroy a CeedQFunction.

User Functions

Parameters

- **qf** – [**inout**] CeedQFunction to destroy

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextCreate**(*Ceed* ceed, *CeedQFunctionContext* *ctx)

Create a CeedQFunctionContext for storing CeedQFunction user context data.

User Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedQFunctionContext will be created
- **ctx** – [**out**] Address of the variable where the newly created CeedQFunctionContext will be stored

Returns
An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextReferenceCopy`**(*CeedQFunctionContext* ctx, *CeedQFunctionContext* *ctx_copy)

Copy the pointer to a CeedQFunctionContext.

Both pointers should be destroyed with *`CeedQFunctionContextDestroy()`*.

Note: If the value of `ctx_copy` passed to this function is non-NULL, then it is assumed that `ctx_copy` is a pointer to a CeedQFunctionContext. This CeedQFunctionContext will be destroyed if `ctx_copy` is the only reference to this CeedQFunctionContext.

User Functions

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext to copy reference to
- **`ctx_copy`** – [**inout**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextSetData`**(*CeedQFunctionContext* ctx, *CeedMemType* mem_type, *CeedCopyMode* copy_mode, size_t size, void *data)

Set the data used by a CeedQFunctionContext, freeing any previously allocated data if applicable.

The backend may copy values to a different memtype, such as during *CeedQFunctionApply()*. See also *CeedQFunctionContextTakeData()*.

User Functions

Parameters

- **`ctx`** – [**inout**] CeedQFunctionContext
- **`mem_type`** – [**in**] Memory type of the data being passed
- **`copy_mode`** – [**in**] Copy mode for the data
- **`size`** – [**in**] Size of data, in bytes
- **`data`** – [**in**] Data to be used

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextTakeData`**(*CeedQFunctionContext* ctx, *CeedMemType* mem_type, void *data)

Take ownership of the data in a CeedQFunctionContext via the specified memory type.

The caller is responsible for managing and freeing the memory.

User Functions

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext to access
- **`mem_type`** – [**in**] Memory type on which to access the data. If the backend uses a different memory type, this will perform a copy.
- **`data`** – [**out**] Data on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextGetData`**(*CeedQFunctionContext* ctx, *CeedMemType* mem_type, void *data)

Get read/write access to a CeedQFunctionContext via the specified memory type.

Restore access with *CeedQFunctionContextRestoreData()*.

User Functions

Note: The *CeedQFunctionContextGetData()* and *CeedQFunctionContextRestoreData()* functions provide access to array pointers in the desired memory space. Pairing get/restore allows the Context to track access.

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext to access
- **`mem_type`** – [**in**] Memory type on which to access the data. If the backend uses a different memory type, this will perform a copy.
- **`data`** – [**out**] Data on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextGetDataRead`**(*CeedQFunctionContext* ctx, *CeedMemType* mem_type, void *data)

Get read only access to a CeedQFunctionContext via the specified memory type.

Restore access with *CeedQFunctionContextRestoreData()*.

User Functions

Note: The *CeedQFunctionContextGetDataRead()* and *CeedQFunctionContextRestoreDataRead()* functions provide access to array pointers in the desired memory space. Pairing get/restore allows the Context to track access.

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext to access
- **`mem_type`** – [**in**] Memory type on which to access the data. If the backend uses a different memory type, this will perform a copy.
- **`data`** – [**out**] Data on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextRestoreData`**(*CeedQFunctionContext* ctx, void *data)

Restore data obtained using *CeedQFunctionContextGetData()*

User Functions

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext to restore
- **`data`** – [**inout**] Data to restore

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRestoreDataRead**(*CeedQFunctionContext* ctx, void *data)

Restore data obtained using *CeedQFunctionContextGetDataRead()*

User Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext to restore
- **data** – [**inout**] Data to restore

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRegisterDouble**(*CeedQFunctionContext* ctx, const char *field_name, size_t field_offset, size_t num_values, const char *field_description)

Register QFunctionContext a field holding a double precision value.

User Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **field_name** – [**in**] Name of field to register
- **field_offset** – [**in**] Offset of field to register
- **num_values** – [**in**] Number of values to register, must be contiguous in memory
- **field_description** – [**in**] Description of field, or NULL for none

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRegisterInt32**(*CeedQFunctionContext* ctx, const char *field_name, size_t field_offset, size_t num_values, const char *field_description)

Register QFunctionContext a field holding a int32 value.

User Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **field_name** – [**in**] Name of field to register
- **field_offset** – [**in**] Offset of field to register
- **num_values** – [**in**] Number of values to register, must be contiguous in memory
- **field_description** – [**in**] Description of field, or NULL for none

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetAllFieldLabels**(*CeedQFunctionContext* ctx, const *CeedContextFieldLabel* **field_labels, *CeedInt* *num_fields)

Get labels for all registered QFunctionContext fields.

User Functions

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext
- **`field_labels`** – [**out**] Variable to hold array of field labels
- **`num_fields`** – [**out**] Length of field descriptions array

Returns

An error code: 0 - success, otherwise - failure

int **`CeedContextFieldLabelGetDescription`**(*CeedContextFieldLabel* label, const char **field_name, size_t *field_offset, size_t *num_values, const char **field_description, CeedContextFieldType *field_type)

Get the descriptive information about a CeedContextFieldLabel.

User Functions

Parameters

- **`label`** – [**in**] CeedContextFieldLabel
- **`field_name`** – [**out**] Name of labeled field
- **`field_offset`** – [**out**] Offset of field registered
- **`num_values`** – [**out**] Number of values registered
- **`field_description`** – [**out**] Description of field, or NULL for none
- **`field_type`** – [**out**] CeedContextFieldType

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextGetContextSize`**(*CeedQFunctionContext* ctx, size_t *ctx_size)

Get data size for a Context.

User Functions

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext
- **`ctx_size`** – [**out**] Variable to store size of context data values

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionContextView`**(*CeedQFunctionContext* ctx, FILE *stream)

View a CeedQFunctionContext.

User Functions

Parameters

- **`ctx`** – [**in**] CeedQFunctionContext to view
- **`stream`** – [**in**] Filestream to write to

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextSetDataDestroy**(*CeedQFunctionContext* ctx, *CeedMemType* f_mem_type, CeedQFunctionContextDataDestroyUser f)

Set additional destroy routine for CeedQFunctionContext user data.

User Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext to set user destroy function
- **f_mem_type** – [**in**] Memory type to use when passing data into `f`
- **f** – [**in**] Additional routine to use to destroy user data

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextDestroy**(*CeedQFunctionContext* *ctx)

Destroy a CeedQFunctionContext.

User Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext to destroy

Returns

An error code: 0 - success, otherwise - failure

Macros

CEED_QFUNCTION(name)

This macro populates the correct function annotations for User QFunction source for code generation backends or populates default values for CPU backends.

It also creates a variable `name_loc` populated with the correct source path for creating the respective User QFunction.

CEED_QFUNCTION_HELPER

This macro populates the correct function annotations for User QFunction helper function source for code generation backends or populates default values for CPU backends.

CEED_Q_VLA

Using VLA syntax to reshape User QFunction inputs and outputs can make user code more readable.

VLA is a C99 feature that is not supported by the C++ dialect used by CUDA. This macro allows users to use the VLA syntax with the CUDA backends.

6.1.6 CeedOperator

A *CeedOperator* defines the finite/spectral element operator associated to a *CeedQFunction*. A *CeedOperator* connects objects of the type *CeedElemRestriction*, *CeedBasis*, and *CeedQFunction*.

6.1.6.1 Discrete operators on user vectors

typedef struct CeedOperator_private ***CeedOperator**

Handle for object describing FE-type operators acting on vectors.

Given an element restriction E, basis evaluator B, and quadrature function f, a CeedOperator expresses operations of the form $E^T B^T f(BEu)$ acting on the vector u.

int **CeedOperatorCreate**(*Ceed* ceed, *CeedQFunction* qf, *CeedQFunction* dqf, *CeedQFunction* dqfT, *CeedOperator* *op)

Create a CeedOperator and associate a CeedQFunction.

A CeedBasis and CeedElemRestriction can be associated with CeedQFunction fields with *CeedOperatorSetField*.

User Functions

Parameters

- **ceed** – **[in]** Ceed object where the CeedOperator will be created
- **qf** – **[in]** QFunction defining the action of the operator at quadrature points
- **dqf** – **[in]** QFunction defining the action of the Jacobian of *qf* (or CEED_QFUNCTION_NONE)
- **dqfT** – **[in]** QFunction defining the action of the transpose of the Jacobian of *qf* (or CEED_QFUNCTION_NONE)
- **op** – **[out]** Address of the variable where the newly created CeedOperator will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedCompositeOperatorCreate**(*Ceed* ceed, *CeedOperator* *op)

Create an operator that composes the action of several operators.

User Functions

Parameters

- **ceed** – **[in]** Ceed object where the CeedOperator will be created
- **op** – **[out]** Address of the variable where the newly created Composite CeedOperator will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorReferenceCopy**(*CeedOperator* op, *CeedOperator* *op_copy)

Copy the pointer to a CeedOperator.

Both pointers should be destroyed with `CeedOperatorDestroy()`.

Note: If the value of `op_copy` passed to this function is non-NULL, then it is assumed that `op_copy` is a pointer to a CeedOperator. This CeedOperator will be destroyed if `op_copy` is the only reference to this CeedOperator.

User Functions

Parameters

- **`op`** – **[in]** CeedOperator to copy reference to
- **`op_copy`** – **[inout]** Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorSetField`**(*CeedOperator* op, const char *field_name, *CeedElemRestriction* r, *CeedBasis* b, *CeedVector* v)

Provide a field to a CeedOperator for use by its CeedQFunction.

This function is used to specify both active and passive fields to a CeedOperator. For passive fields, a vector

- v must be provided. Passive fields can inputs or outputs (updated in-place when operator is applied).

Active fields must be specified using this function, but their data (in a CeedVector) is passed in *CeedOperatorApply()*. There can be at most one active input CeedVector and at most one active output CeedVector passed to *CeedOperatorApply()*.

The number of quadrature points must agree across all points. When using CEED_BASIS_NONE, the number of quadrature points is determined by the element size of r.

User Functions

Parameters

- **`op`** – **[inout]** CeedOperator on which to provide the field
- **`field_name`** – **[in]** Name of the field (to be matched with the name used by CeedQFunction)
- **`r`** – **[in]** CeedElemRestriction
- **`b`** – **[in]** CeedBasis in which the field resides or CEED_BASIS_NONE if collocated with quadrature points
- **`v`** – **[in]** CeedVector to be used by CeedOperator or CEED_VECTOR_ACTIVE if field is active or CEED_VECTOR_NONE if using *CEED_EVAL_WEIGHT* in the QFunction

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorGetFields`**(*CeedOperator* op, *CeedInt* *num_input_fields, *CeedOperatorField* **input_fields, *CeedInt* *num_output_fields, *CeedOperatorField* **output_fields)

Get the CeedOperatorFields of a CeedOperator.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

Advanced Functions

Parameters

- **`op`** – **[in]** CeedOperator

- **num_input_fields** – [**out**] Variable to store number of input fields
- **input_fields** – [**out**] Variable to store input_fields
- **num_output_fields** – [**out**] Variable to store number of output fields
- **output_fields** – [**out**] Variable to store output_fields

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorGetFieldByName**(*CeedOperator* op, const char *field_name, *CeedOperatorField* *op_field)

Get a CeedOperatorField of an CeedOperator from its name.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

Advanced Functions

Parameters

- **op** – [**in**] CeedOperator
- **field_name** – [**in**] Name of desired CeedOperatorField
- **op_field** – [**out**] CeedOperatorField corresponding to the name

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorFieldGetName**(*CeedOperatorField* op_field, char **field_name)

Get the name of a CeedOperatorField.

Advanced Functions

Parameters

- **op_field** – [**in**] CeedOperatorField
- **field_name** – [**out**] Variable to store the field name

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorFieldGetElemRestriction**(*CeedOperatorField* op_field, *CeedElemRestriction* *rstr)

Get the CeedElemRestriction of a CeedOperatorField.

Advanced Functions

Parameters

- **op_field** – [**in**] CeedOperatorField
- **rstr** – [**out**] Variable to store CeedElemRestriction

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorFieldGetBasis**(*CeedOperatorField* op_field, *CeedBasis* *basis)

Get the CeedBasis of a CeedOperatorField.

Advanced Functions

Parameters

- **op_field** – [**in**] CeedOperatorField

- **basis** – [**out**] Variable to store CeedBasis

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorFieldGetVector**(*CeedOperatorField* op_field, *CeedVector* *vec)

Get the CeedVector of a CeedOperatorField.

Advanced Functions

Parameters

- **op_field** – [**in**] CeedOperatorField
- **vec** – [**out**] Variable to store CeedVector

Returns
An error code: 0 - success, otherwise - failure

int **CeedCompositeOperatorAddSub**(*CeedOperator* composite_op, *CeedOperator* sub_op)

Add a sub-operator to a composite CeedOperator.

User Functions

Parameters

- **composite_op** – [**inout**] Composite CeedOperator
- **sub_op** – [**in**] Sub-operator CeedOperator

Returns
An error code: 0 - success, otherwise - failure

int **CeedCompositeOperatorGetNumSub**(*CeedOperator* op, *CeedInt* *num_suboperators)

Get the number of sub_operators associated with a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [**in**] CeedOperator
- **num_suboperators** – [**out**] Variable to store number of sub_operators

Returns
An error code: 0 - success, otherwise - failure

int **CeedCompositeOperatorGetSubList**(*CeedOperator* op, *CeedOperator* **sub_operators)

Get the list of sub_operators associated with a CeedOperator.

Backend Developer Functions

Parameters

- **op** – CeedOperator
- **sub_operators** – [**out**] Variable to store list of sub_operators

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorCheckReady**(*CeedOperator* op)

Check if a CeedOperator is ready to be used.

User Functions

Parameters

- **op** – [**in**] CeedOperator to check

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorGetActiveVectorLengths**(*CeedOperator* op, CeedSize *input_size, CeedSize *output_size)

Get vector lengths for the active input and/or output vectors of a CeedOperator.

Note: Lengths of -1 indicate that the CeedOperator does not have an active input and/or output.

User Functions

Parameters

- **op** – [**in**] CeedOperator
- **input_size** – [**out**] Variable to store active input vector length, or NULL
- **output_size** – [**out**] Variable to store active output vector length, or NULL

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorSetQFunctionAssemblyReuse**(*CeedOperator* op, bool reuse_assembly_data)

Set reuse of CeedQFunction data in CeedOperatorLinearAssemble* functions.

When `reuse_assembly_data = false` (default), the CeedQFunction associated with this CeedOperator is re-assembled every time a `CeedOperatorLinearAssemble*` function is called. When `reuse_assembly_data = true`, the CeedQFunction associated with this CeedOperator is reused between calls to `CeedOperatorSetQFunctionAssemblyDataUpdated`.

Advanced Functions

Parameters

- **op** – [**in**] CeedOperator
- **reuse_assembly_data** – [**in**] Boolean flag setting assembly data reuse

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorSetQFunctionAssemblyDataUpdateNeeded**(*CeedOperator* op, bool needs_data_update)

Mark CeedQFunction data as updated and the CeedQFunction as requiring re-assembly.

Advanced Functions

Parameters

- **op** – [**in**] CeedOperator
- **needs_data_update** – [**in**] Boolean flag setting assembly data reuse

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorSetName**(*CeedOperator* op, const char *name)

Set name of CeedOperator for CeedOperatorView output.

User Functions

Parameters

- **op** – [**inout**] CeedOperator

- **name** – [**in**] Name to set, or NULL to remove previously set name

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorView**(*CeedOperator* op, FILE *stream)

View a CeedOperator.

User Functions

Parameters

- **op** – [**in**] CeedOperator to view
- **stream** – [**in**] Stream to write; typically stdout/stderr or a file

Returns
Error code: 0 - success, otherwise - failure

int **CeedOperatorGetCeed**(*CeedOperator* op, *Ceed* *ceed)

Get the Ceed associated with a CeedOperator.

Advanced Functions

Parameters

- **op** – [**in**] CeedOperator
- **ceed** – [**out**] Variable to store Ceed

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorGetNumElements**(*CeedOperator* op, *CeedInt* *num_elem)

Get the number of elements associated with a CeedOperator.

Advanced Functions

Parameters

- **op** – [**in**] CeedOperator
- **num_elem** – [**out**] Variable to store number of elements

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorGetNumQuadraturePoints**(*CeedOperator* op, *CeedInt* *num_qpts)

Get the number of quadrature points associated with a CeedOperator.

Advanced Functions

Parameters

- **op** – [**in**] CeedOperator
- **num_qpts** – [**out**] Variable to store vector number of quadrature points

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorGetFlopsEstimate**(*CeedOperator* op, CeedSize *flops)

Estimate number of FLOPs required to apply CeedOperator on the active vector.

Backend Developer Functions

Parameters

- **op** – [in] CeedOperator to estimate FLOPs for
- **flops** – [out] Address of variable to hold FLOPs estimate

int **CeedOperatorGetContext**(*CeedOperator* op, *CeedQFunctionContext* *ctx)

Get CeedQFunction global context for a CeedOperator.

The caller is responsible for destroying `ctx` returned from this function via *`CeedQFunctionContextDestroy()`*.

Note: If the value of `ctx` passed into this function is non-NULL, then it is assumed that `ctx` is a pointer to a CeedQFunctionContext. This CeedQFunctionContext will be destroyed if `ctx` is the only reference to this CeedQFunctionContext.

Advanced Functions

Parameters

- **op** – [in] CeedOperator
- **ctx** – [out] Variable to store CeedQFunctionContext

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetContextFieldLabel**(*CeedOperator* op, const char *field_name, *CeedContextFieldLabel* *field_label)

Get label for a registered QFunctionContext field, or `NULL` if no field has been registered with this `field_name`.

Fields are registered via `CeedQFunctionContextRegister*()` functions (eg. *`CeedQFunctionContextRegisterDouble()`*).

User Functions

Parameters

- **op** – [in] CeedOperator
- **field_name** – [in] Name of field to retrieve label
- **field_label** – [out] Variable to field label

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorSetContextDouble**(*CeedOperator* op, *CeedContextFieldLabel* field_label, double *values)

Set QFunctionContext field holding double precision values.

For composite operators, the values are set in all sub-operator QFunctionContexts that have a matching `field_name`.

User Functions

Parameters

- **op** – [inout] CeedOperator
- **field_label** – [in] Label of field to set
- **values** – [in] Values to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetContextDoubleRead**(*CeedOperator* op, *CeedContextFieldLabel* field_label, size_t *num_values, const double **values)

Get QFunctionContext field holding double precision values, read-only.

For composite operators, the values correspond to the first sub-operator QFunctionContexts that has a matching `field_name`.

User Functions

Parameters

- **op** – [**in**] CeedOperator
- **field_label** – [**in**] Label of field to get
- **num_values** – [**out**] Number of values in the field label
- **values** – [**out**] Pointer to context values

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorRestoreContextDoubleRead**(*CeedOperator* op, *CeedContextFieldLabel* field_label, const double **values)

Restore QFunctionContext field holding double precision values, read-only.

User Functions

Parameters

- **op** – [**in**] CeedOperator
- **field_label** – [**in**] Label of field to restore
- **values** – [**out**] Pointer to context values

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorSetContextInt32**(*CeedOperator* op, *CeedContextFieldLabel* field_label, int *values)

Set QFunctionContext field holding int32 values.

For composite operators, the values are set in all sub-operator QFunctionContexts that have a matching `field_name`.

User Functions

Parameters

- **op** – [**inout**] CeedOperator
- **field_label** – [**in**] Label of field to set
- **values** – [**in**] Values to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorGetContextInt32Read**(*CeedOperator* op, *CeedContextFieldLabel* field_label, size_t *num_values, const int **values)

Get QFunctionContext field holding int32 values, read-only.

For composite operators, the values correspond to the first sub-operator QFunctionContexts that has a matching `field_name`.

User Functions

Parameters

- **`op`** – **[in]** CeedOperator
- **`field_label`** – **[in]** Label of field to get
- **`num_values`** – **[out]** Number of int32 values in `values`
- **`values`** – **[out]** Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorRestoreContextInt32Read`**(*CeedOperator* op, *CeedContextFieldLabel* field_label, const int **values)

Restore QFunctionContext field holding int32 values, read-only.

User Functions

Parameters

- **`op`** – **[in]** CeedOperator
- **`field_label`** – **[in]** Label of field to get
- **`values`** – **[out]** Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorApply`**(*CeedOperator* op, *CeedVector* in, *CeedVector* out, *CeedRequest* *request)

Apply CeedOperator to a vector.

This computes the action of the operator on the specified (active) input, yielding its (active) output. All inputs and outputs must be specified using *CeedOperatorSetField()*.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **`op`** – **[in]** CeedOperator to apply
- **`in`** – **[in]** CeedVector containing input state or CEED_VECTOR_NONE if there are no active inputs
- **`out`** – **[out]** CeedVector to store result of applying operator (must be distinct from *in*) or CEED_VECTOR_NONE if there are no active outputs
- **`request`** – **[in]** Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorApplyAdd`**(*CeedOperator* op, *CeedVector* in, *CeedVector* out, *CeedRequest* *request)

Apply CeedOperator to a vector and add result to output vector.

This computes the action of the operator on the specified (active) input, yielding its (active) output. All inputs and outputs must be specified using *CeedOperatorSetField()*.

User Functions

Parameters

- **`op`** – **[in]** CeedOperator to apply

- **in** – [in] CeedVector containing input state or NULL if there are no active inputs
- **out** – [out] CeedVector to sum in result of applying operator (must be distinct from *in*) or NULL if there are no active outputs
- **request** – [in] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorDestroy**(*CeedOperator* *op)

Destroy a CeedOperator.

User Functions

Parameters

- **op** – [inout] CeedOperator to destroy

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorLinearAssembleQFunction**(*CeedOperator* op, *CeedVector* *assembled, *CeedElemRestriction* *rstr, *CeedRequest* *request)

Assemble a linear CeedQFunction associated with a CeedOperator.

This returns a CeedVector containing a matrix at each quadrature point providing the action of the CeedQFunction associated with the CeedOperator. The vector `assembled` is of shape `[num_elements, num_input_fields, num_output_fields, num_quad_points]` and contains column-major matrices representing the action of the CeedQFunction for a corresponding quadrature point on an element.

Inputs and outputs are in the order provided by the user when adding CeedOperator fields. For example, a CeedQFunction with inputs 'u' and 'gradu' and outputs 'gradv' and 'v', provided in that order, would result in an assembled QFunction that consists of (1 + dim) x (dim + 1) matrices at each quadrature point acting on the input [u, du_0, du_1] and producing the output [dv_0, dv_1, v].

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [in] CeedOperator to assemble CeedQFunction
- **assembled** – [out] CeedVector to store assembled CeedQFunction at quadrature points
- **rstr** – [out] CeedElemRestriction for CeedVector containing assembled CeedQFunction
- **request** – [in] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorLinearAssembleQFunctionBuildOrUpdate**(*CeedOperator* op, *CeedVector* *assembled, *CeedElemRestriction* *rstr, *CeedRequest* *request)

Assemble CeedQFunction and store result internally.

Return copied references of stored data to the caller. Caller is responsible for ownership and destruction of the copied references. See also *CeedOperatorLinearAssembleQFunction*

Note: If the value of `assembled` or `rstr` passed to this function are non-NULL, then it is assumed that they hold valid pointers. These objects will be destroyed if `*assembled` or `*rstr` is the only reference to the object.

User Functions

Parameters

- **`op`** – [**in**] CeedOperator to assemble CeedQFunction
- **`assembled`** – [**out**] CeedVector to store assembled CeedQFunction at quadrature points
- **`rstr`** – [**out**] CeedElemRestriction for CeedVector containing assembledCeedQFunction
- **`request`** – [**in**] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorLinearAssembleDiagonal`**(*CeedOperator* op, *CeedVector* assembled, *CeedRequest* *request)

Assemble the diagonal of a square linear CeedOperator.

This overwrites a CeedVector with the diagonal of a linear CeedOperator.

Note: Currently only non-composite CeedOperators with a single field and composite CeedOperators with single field sub-operators are supported.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **`op`** – [**in**] CeedOperator to assemble CeedQFunction
- **`assembled`** – [**out**] CeedVector to store assembled CeedOperator diagonal
- **`request`** – [**in**] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorLinearAssembleAddDiagonal`**(*CeedOperator* op, *CeedVector* assembled, *CeedRequest* *request)

Assemble the diagonal of a square linear CeedOperator.

This sums into a CeedVector the diagonal of a linear CeedOperator.

Note: Currently only non-composite CeedOperators with a single field and composite CeedOperators with single field sub-operators are supported.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **`op`** – [**in**] CeedOperator to assemble CeedQFunction

- **assembled** – [**out**] CeedVector to store assembled CeedOperator diagonal
- **request** – [**in**] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorLinearAssemblePointBlockDiagonalSymbolic**(*CeedOperator* op, CeedSize *num_entries, *CeedInt* **rows, *CeedInt* **cols)

Fully assemble the point-block diagonal pattern of a linear operator.

Expected to be used in conjunction with *CeedOperatorLinearAssemblePointBlockDiagonal()*.

The assembly routines use coordinate format, with `num_entries` tuples of the form (i, j, value) which indicate that value should be added to the matrix in entry (i, j). Note that the (i, j) pairs are unique. This function returns the number of entries and their (i, j) locations, while *CeedOperatorLinearAssemblePointBlockDiagonal()* provides the values in the same ordering.

This will generally be slow unless your operator is low-order.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] CeedOperator to assemble
- **num_entries** – [**out**] Number of entries in coordinate nonzero pattern
- **rows** – [**out**] Row number for each entry
- **cols** – [**out**] Column number for each entry

int **CeedOperatorLinearAssemblePointBlockDiagonal**(*CeedOperator* op, *CeedVector* assembled, *CeedRequest* *request)

Assemble the point block diagonal of a square linear CeedOperator.

This overwrites a CeedVector with the point block diagonal of a linear CeedOperator.

Note: Currently only non-composite CeedOperators with a single field and composite CeedOperators with single field sub-operators are supported.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] CeedOperator to assemble CeedQFunction
- **assembled** – [**out**] CeedVector to store assembled CeedOperator point block diagonal, provided in row-major form with an *num_comp* * *num_comp* block at each node. The dimensions of this vector are derived from the active vector for the CeedOperator. The array has shape [nodes, component out, component in].
- **request** – [**in**] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorLinearAssembleAddPointBlockDiagonal**(*CeedOperator* op, *CeedVector* assembled, *CeedRequest* *request)

Assemble the point block diagonal of a square linear CeedOperator.

This sums into a CeedVector with the point block diagonal of a linear CeedOperator.

Note: Currently only non-composite CeedOperators with a single field and composite CeedOperators with single field sub-operators are supported.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] CeedOperator to assemble CeedQFunction
- **assembled** – [**out**] CeedVector to store assembled CeedOperator point block diagonal, provided in row-major form with an *num_comp* * *num_comp* block at each node. The dimensions of this vector are derived from the active vector for the CeedOperator. The array has shape [nodes, component out, component in].
- **request** – [**in**] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorLinearAssembleSymbolic**(*CeedOperator* op, CeedSize *num_entries, *CeedInt* **rows, *CeedInt* **cols)

Fully assemble the nonzero pattern of a linear operator.

Expected to be used in conjunction with *CeedOperatorLinearAssemble()*.

The assembly routines use coordinate format, with num_entries tuples of the form (i, j, value) which indicate that value should be added to the matrix in entry (i, j). Note that the (i, j) pairs are not unique and may repeat. This function returns the number of entries and their (i, j) locations, while *CeedOperatorLinearAssemble()* provides the values in the same ordering.

This will generally be slow unless your operator is low-order.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] CeedOperator to assemble
- **num_entries** – [**out**] Number of entries in coordinate nonzero pattern
- **rows** – [**out**] Row number for each entry
- **cols** – [**out**] Column number for each entry

int **CeedOperatorLinearAssemble**(*CeedOperator* op, *CeedVector* values)

Fully assemble the nonzero entries of a linear operator.

Expected to be used in conjunction with *CeedOperatorLinearAssembleSymbolic()*.

The assembly routines use coordinate format, with num_entries tuples of the form (i, j, value) which indicate that value should be added to the matrix in entry (i, j). Note that the (i, j) pairs are not unique and may repeat. This function returns the values of the nonzero entries to be added, their (i, j) locations are provided by *CeedOperatorLinearAssembleSymbolic()*

This will generally be slow unless your operator is low-order.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] CeedOperator to assemble
- **values** – [**out**] Values to assemble into matrix

int **CeedCompositeOperatorGetMultiplicity**(*CeedOperator* op, *CeedInt* num_skip_indices, *CeedInt* *skip_indices, *CeedVector* mult)

Get the multiplicity of nodes across suboperators in a composite CeedOperator.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] Composite CeedOperator
- **num_skip_indices** – [**in**] Number of suboperators to skip
- **skip_indices** – [**in**] Array of indices of suboperators to skip
- **mult** – [**out**] Vector to store multiplicity (of size l_size)

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorMultigridLevelCreate**(*CeedOperator* op_fine, *CeedVector* p_mult_fine, *CeedElemRestriction* rstr_coarse, *CeedBasis* basis_coarse, *CeedOperator* *op_coarse, *CeedOperator* *op_prolong, *CeedOperator* *op_restrict)

Create a multigrid coarse operator and level transfer operators for a CeedOperator, creating the prolongation basis from the fine and coarse grid interpolation.

Note: Calling this function asserts that setup is complete and sets all four CeedOperators as immutable.

User Functions

Parameters

- **op_fine** – [**in**] Fine grid operator
- **p_mult_fine** – [**in**] L-vector multiplicity in parallel gather/scatter, or NULL if not creating prolongation/restriction operators
- **rstr_coarse** – [**in**] Coarse grid restriction
- **basis_coarse** – [**in**] Coarse grid active vector basis
- **op_coarse** – [**out**] Coarse grid operator
- **op_prolong** – [**out**] Coarse to fine operator, or NULL
- **op_restrict** – [**out**] Fine to coarse operator, or NULL

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorMultigridLevelCreateTensorH1**(*CeedOperator* op_fine, *CeedVector* p_mult_fine, *CeedElemRestriction* rstr_coarse, *CeedBasis* basis_coarse, const *CeedScalar* *interp_c_to_f, *CeedOperator* *op_coarse, *CeedOperator* *op_prolong, *CeedOperator* *op_restrict)

Create a multigrid coarse operator and level transfer operators for a CeedOperator with a tensor basis for the active basis.

Note: Calling this function asserts that setup is complete and sets all four CeedOperators as immutable.

User Functions

Parameters

- **op_fine** – [**in**] Fine grid operator
- **p_mult_fine** – [**in**] L-vector multiplicity in parallel gather/scatter, or NULL if not creating prolongation/restriction operators
- **rstr_coarse** – [**in**] Coarse grid restriction
- **basis_coarse** – [**in**] Coarse grid active vector basis
- **interp_c_to_f** – [**in**] Matrix for coarse to fine interpolation, or NULL if not creating prolongation/restriction operators
- **op_coarse** – [**out**] Coarse grid operator
- **op_prolong** – [**out**] Coarse to fine operator, or NULL
- **op_restrict** – [**out**] Fine to coarse operator, or NULL

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorMultigridLevelCreateH1**(*CeedOperator* op_fine, *CeedVector* p_mult_fine, *CeedElemRestriction* rstr_coarse, *CeedBasis* basis_coarse, const *CeedScalar* *interp_c_to_f, *CeedOperator* *op_coarse, *CeedOperator* *op_prolong, *CeedOperator* *op_restrict)

Create a multigrid coarse operator and level transfer operators for a CeedOperator with a non-tensor basis for the active vector.

Note: Calling this function asserts that setup is complete and sets all four CeedOperators as immutable.

User Functions

Parameters

- **op_fine** – [**in**] Fine grid operator
- **p_mult_fine** – [**in**] L-vector multiplicity in parallel gather/scatter, or NULL if not creating prolongation/restriction operators
- **rstr_coarse** – [**in**] Coarse grid restriction
- **basis_coarse** – [**in**] Coarse grid active vector basis
- **interp_c_to_f** – [**in**] Matrix for coarse to fine interpolation, or NULL if not creating prolongation/restriction operators
- **op_coarse** – [**out**] Coarse grid operator
- **op_prolong** – [**out**] Coarse to fine operator, or NULL

- **op_restrict** – [**out**] Fine to coarse operator, or NULL

Returns
An error code: 0 - success, otherwise - failure

int **CeedOperatorCreateFDMElementInverse**(*CeedOperator* op, *CeedOperator* *fdm_inv, *CeedRequest* *request)

Build a FDM based approximate inverse for each element for a CeedOperator.

This returns a CeedOperator and CeedVector to apply a Fast Diagonalization Method based approximate inverse. This function obtains the simultaneous diagonalization for the 1D mass and Laplacian operators, $M = V^T V, K = V^T S V$. The assembled QFunction is used to modify the eigenvalues from simultaneous diagonalization and obtain an approximate inverse of the form $V^T \hat{S} V$. The CeedOperator must be linear and non-composite. The associated CeedQFunction must therefore also be linear.

Note: Calling this function asserts that setup is complete and sets the CeedOperator as immutable.

User Functions

Parameters

- **op** – [**in**] CeedOperator to create element inverses
- **fdm_inv** – [**out**] CeedOperator to apply the action of a FDM based inverse for each element
- **request** – [**in**] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

Returns
An error code: 0 - success, otherwise - failure

6.2 Backend API

These functions are intended to be used by backend developers of libCEED and can generally be found in *ceed-backend.h*.

6.2.1 Ceed

bool **CeedDebugFlag**(const *Ceed* ceed)

Return value of CEED_DEBUG environment variable.

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed context

Returns
boolean value: true - debugging mode enabled false - debugging mode disabled

bool **CeedDebugFlagEnv**(void)

Return value of CEED_DEBUG environment variable.

Backend Developer Functions

Returns
boolean value: true - debugging mode enabled false - debugging mode disabled

void **CeedDebugImpl256**(const unsigned char color, const char *format, ...)

Print debugging information in color.

Backend Developer Functions

Parameters

- **color** – Color to print
- **format** – Printing format

int **CeedMallocArray**(size_t n, size_t unit, void *p)

Allocate an array on the host; use CeedMalloc()

Memory usage can be tracked by the library. This ensures sufficient alignment for vectorization and should be used for large allocations.

Backend Developer Functions

See also:

CeedFree()

Parameters

- **n** – [**in**] Number of units to allocate
- **unit** – [**in**] Size of each unit
- **p** – [**out**] Address of pointer to hold the result.

Returns

An error code: 0 - success, otherwise - failure

int **CeedCallocArray**(size_t n, size_t unit, void *p)

Allocate a cleared (zeroed) array on the host; use CeedCalloc()

Memory usage can be tracked by the library.

Backend Developer Functions

See also:

CeedFree()

Parameters

- **n** – [**in**] Number of units to allocate
- **unit** – [**in**] Size of each unit
- **p** – [**out**] Address of pointer to hold the result.

Returns

An error code: 0 - success, otherwise - failure

int **CeedReallocArray**(size_t n, size_t unit, void *p)

Reallocate an array on the host; use CeedRealloc()

Memory usage can be tracked by the library.

Backend Developer Functions

See also:

CeedFree()

Parameters

- **n** – [**in**] Number of units to allocate
- **unit** – [**in**] Size of each unit
- **p** – [**out**] Address of pointer to hold the result.

Returns

An error code: 0 - success, otherwise - failure

int **CeedStringAllocCopy**(const char *source, char **copy)

Allocate a cleared string buffer on the host.

Memory usage can be tracked by the library.

Backend Developer Functions

See also:

CeedFree()

Parameters

- **source** – [**in**] Pointer to string to be copied
- **copy** – [**out**] Pointer to variable to hold newly allocated string copy

Returns

An error code: 0 - success, otherwise - failure

int **CeedFree**(void *p)

Free memory allocated using CeedMalloc() or CeedCalloc()

Parameters

- **p** – [**inout**] address of pointer to memory. This argument is of type void* to avoid needing a cast, but is the address of the pointer (which is zeroed) rather than the pointer.

int **CeedRegister**(const char *prefix, int (*init)(const char*, *Ceed*), unsigned int priority)

Register a Ceed backend.

Backend Developer Functions

Parameters

- **prefix** – [**in**] Prefix of resources for this backend to respond to. For example, the reference backend responds to "/cpu/self".
- **init** – [**in**] Initialization function called by *CeedInit()* when the backend is selected to drive the requested resource.
- **priority** – [**in**] Integer priority. Lower values are preferred in case the resource requested by *CeedInit()* has non-unique best prefix match.

Returns

An error code: 0 - success, otherwise - failure

int **CeedIsDebug**(*Ceed* ceed, bool *is_debug)

Return debugging status flag.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed context to get debugging flag
- **`is_debug`** – [**out**] Variable to store debugging flag

Returns

An error code: 0 - success, otherwise - failure

int **`CeedGetResourceRoot`**(*Ceed* ceed, const char *resource, const char *delineator, char **resource_root)

Get the root of the requested resource.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed context to get resource name of
- **`resource`** – [**in**] ull user specified resource
- **`delineator`** – [**in**] Delineator to break resource_root and resource_spec
- **`resource_root`** – [**out**] Variable to store resource root

Returns

An error code: 0 - success, otherwise - failure

int **`CeedGetParent`**(*Ceed* ceed, *Ceed* *parent)

Retrieve a parent Ceed context.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed context to retrieve parent of
- **`parent`** – [**out**] Address to save the parent to

Returns

An error code: 0 - success, otherwise - failure

int **`CeedGetDelegate`**(*Ceed* ceed, *Ceed* *delegate)

Retrieve a delegate Ceed context.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed context to retrieve delegate of
- **`delegate`** – [**out**] Address to save the delegate to

Returns

An error code: 0 - success, otherwise - failure

int **`CeedSetDelegate`**(*Ceed* ceed, *Ceed* delegate)

Set a delegate Ceed context.

This function allows a Ceed context to set a delegate Ceed context. All backend implementations default to the delegate Ceed context, unless overridden.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed context to set delegate of

- **delegate** – [**out**] Address to set the delegate to

Returns
An error code: 0 - success, otherwise - failure

int **CeedGetObjectDelegate**(*Ceed* ceed, *Ceed* *delegate, const char *obj_name)

Retrieve a delegate Ceed context for a specific object type.

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed context to retrieve delegate of
- **delegate** – [**out**] Address to save the delegate to
- **obj_name** – [**in**] Name of the object type to retrieve delegate for

Returns
An error code: 0 - success, otherwise - failure

int **CeedSetObjectDelegate**(*Ceed* ceed, *Ceed* delegate, const char *obj_name)

Set a delegate Ceed context for a specific object type.

This function allows a Ceed context to set a delegate Ceed context for a given type of Ceed object. All backend implementations default to the delegate Ceed context for this object. For example, CeedSetObjectDelegate(ceed, delegate, "Basis") uses delegate implementations for all CeedBasis backend functions.

Backend Developer Functions

Parameters

- **ceed** – [**inout**] Ceed context to set delegate of
- **delegate** – [**in**] Ceed context to use for delegation
- **obj_name** – [**in**] Name of the object type to set delegate for

Returns
An error code: 0 - success, otherwise - failure

int **CeedGetOperatorFallbackResource**(*Ceed* ceed, const char **resource)

Get the fallback resource for CeedOperators.

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed context
- **resource** – [**out**] Variable to store fallback resource

Returns
An error code: 0 - success, otherwise - failure

int **CeedGetOperatorFallbackCeed**(*Ceed* ceed, *Ceed* *fallback_ceed)

Get the fallback Ceed for CeedOperators.

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed context
- **fallback_ceed** – [**out**] Variable to store fallback Ceed

Returns
An error code: 0 - success, otherwise - failure

int **CeedSetOperatorFallbackResource**(*Ceed* ceed, const char *resource)

Set the fallback resource for CeedOperators.

The current resource, if any, is freed by calling this function. This string is freed upon the destruction of the Ceed context.

Backend Developer Functions

Parameters

- **ceed** – [**inout**] Ceed context
- **resource** – [**in**] Fallback resource to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedSetDeterministic**(*Ceed* ceed, bool is_deterministic)

Flag Ceed context as deterministic.

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed to flag as deterministic
- **is_deterministic** – [**out**] Deterministic status to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedSetBackendFunction**(*Ceed* ceed, const char *type, void *object, const char *func_name, int (*f)())

Set a backend function.

This function is used for a backend to set the function associated with the Ceed objects. For example, CeedSetBackendFunction(ceed, "Ceed", ceed, "VectorCreate", BackendVectorCreate) sets the backend implementation of 'CeedVectorCreate' and CeedSetBackendFunction(ceed, "Basis", basis, "Apply", BackendBasisApply) sets the backend implementation of 'CeedBasisApply'. Note, the prefix 'Ceed' is not required for the object type ("Basis" vs "CeedBasis").

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed context for error handling
- **type** – [**in**] Type of Ceed object to set function for
- **object** – [**out**] Ceed object to set function for
- **func_name** – [**in**] Name of function to set
- **f** – [**in**] Function to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedGetData**(*Ceed* ceed, void *data)

Retrieve backend data for a Ceed context.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed context to retrieve data of
- **`data`** – [**out**] Address to save data to

Returns
An error code: 0 - success, otherwise - failure

int **`CeedSetData`**(*Ceed* ceed, void *data)

Set backend data for a Ceed context.

Backend Developer Functions

Parameters

- **`ceed`** – [**inout**] Ceed context to set data of
- **`data`** – [**in**] Address of data to set

Returns
An error code: 0 - success, otherwise - failure

int **`CeedReference`**(*Ceed* ceed)

Increment the reference counter for a Ceed context.

Backend Developer Functions

Parameters

- **`ceed`** – [**inout**] Ceed context to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

6.2.1.1 Macros

`CeedDebug256`(ceed, color, ...)

Print debugging information in color.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed
- **`color`** – [**in**] Color to print with

`CeedDebug`(ceed, ...)

Print debugging information to terminal.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed

`CeedDebugEnv256`(color, ...)

Print debugging information in color without Ceed to reference.

Backend Developer Functions

Parameters

- **`color`** – [**in**] Color to print with

CeedDebugEnv(...)

Print debugging information to terminal without Ceed to reference.

Backend Developer Functions

6.2.1.2 Typedefs and Enumerations

enum **CeedDebugColor**

This enum supples common colors for CeedDebug256 debugging output.

Set the environment variable `CEED_DEBUG = 1` to activate debugging output.

Backend Developer Functions

Values:

enumerator **CEED_DEBUG_COLOR_SUCCESS**

Success color.

enumerator **CEED_DEBUG_COLOR_WARNING**

Warning color.

enumerator **CEED_DEBUG_COLOR_ERROR**

Error color.

enumerator **CEED_DEBUG_COLOR_NONE**

Use native terminal coloring.

6.2.2 CeedVector

int **CeedVectorHasValidArray**(*CeedVector* vec, bool *has_valid_array)

Check for valid data in a CeedVector.

Backend Developer Functions

Parameters

- **vec** – [**in**] CeedVector to check validity
- **has_valid_array** – [**out**] Variable to store validity

Returns

An error code: 0 - success, otherwise - failure

int **CeedVectorHasBorrowedArrayOfType**(*CeedVector* vec, *CeedMemType* mem_type, bool *has_borrowed_array_of_type)

Check for borrowed array of a specific CeedMemType in a CeedVector.

Backend Developer Functions

Parameters

- **vec** – [**in**] CeedVector to check
- **mem_type** – [**in**] Memory type to check

- **has_borrowed_array_of_type** – [**out**] Variable to store result

Returns
An error code: 0 - success, otherwise - failure

int **CeedVectorGetState**(*CeedVector* vec, uint64_t *state)

Get the state of a CeedVector.

Backend Developer Functions

Parameters

- **vec** – [**in**] CeedVector to retrieve state
- **state** – [**out**] Variable to store state

Returns
An error code: 0 - success, otherwise - failure

int **CeedVectorGetData**(*CeedVector* vec, void *data)

Get the backend data of a CeedVector.

Backend Developer Functions

Parameters

- **vec** – [**in**] CeedVector to retrieve state
- **data** – [**out**] Variable to store data

Returns
An error code: 0 - success, otherwise - failure

int **CeedVectorSetData**(*CeedVector* vec, void *data)

Set the backend data of a CeedVector.

Backend Developer Functions

Parameters

- **vec** – [**inout**] CeedVector to retrieve state
- **data** – [**in**] Data to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedVectorReference**(*CeedVector* vec)

Increment the reference counter for a CeedVector.

Backend Developer Functions

Parameters

- **vec** – [**inout**] CeedVector to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

6.2.3 CeedElemRestriction

int **CeedElemRestrictionGetType**(*CeedElemRestriction* rstr, CeedRestrictionType *rstr_type)

Get the type of a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **rstr_type** – [**out**] Variable to store restriction type

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionIsStrided**(*CeedElemRestriction* rstr, bool *is_strided)

Get the strided status of a CeedElemRestriction.

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **is_strided** – [**out**] Variable to store strided status, 1 if strided else 0

int **CeedElemRestrictionIsPoints**(*CeedElemRestriction* rstr, bool *is_points)

Get the points status of a CeedElemRestriction.

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **is_points** – [**out**] Variable to store points status, 1 if points else 0

int **CeedElemRestrictionGetStrides**(*CeedElemRestriction* rstr, *CeedInt* (*strides)[3])

Get the strides of a strided CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **strides** – [**out**] Variable to store strides array

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionHasBackendStrides**(*CeedElemRestriction* rstr, bool *has_backend_strides)

Get the backend stride status of a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **has_backend_strides** – [**out**] Variable to store stride status

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetOffsets**(*CeedElemRestriction* rstr, *CeedMemType* mem_type, const *CeedInt* **offsets)

Get read-only access to a CeedElemRestriction offsets array by memtype.

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction to retrieve offsets
- **mem_type** – [**in**] Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy (possibly cached).
- **offsets** – [**out**] Array on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionRestoreOffsets**(*CeedElemRestriction* rstr, const *CeedInt* **offsets)

Restore an offsets array obtained using *CeedElemRestrictionGetOffsets()*

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction to restore
- **offsets** – [**in**] Array of offset data

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetOrientations**(*CeedElemRestriction* rstr, *CeedMemType* mem_type, const bool **orients)

Get read-only access to a CeedElemRestriction orientations array by memtype.

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction to retrieve orientations
- **mem_type** – [**in**] Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy (possibly cached).
- **orients** – [**out**] Array on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionRestoreOrientations**(*CeedElemRestriction* rstr, const bool **orients)

Restore an orientations array obtained using *CeedElemRestrictionGetOrientations()*

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction to restore
- **orients** – [**in**] Array of orientation data

Returns

An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetCurlOrientations**(*CeedElemRestriction* rstr, *CeedMemType* mem_type, const CeedInt8 **curl_orients)

Get read-only access to a CeedElemRestriction curl-conforming orientations array by memtype.

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction to retrieve curl-conforming orientations
- **mem_type** – [**in**] Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy (possibly cached).
- **curl_orients** – [**out**] Array on memory type mem_type

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionRestoreCurlOrientations**(*CeedElemRestriction* rstr, const CeedInt8 **curl_orients)

Restore an orientations array obtained using *CeedElemRestrictionGetCurlOrientations()*

User Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction to restore
- **curl_orients** – [**in**] Array of orientation data

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetELayout**(*CeedElemRestriction* rstr, *CeedInt* (*layout)[3])

Get the E-vector layout of a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **layout** – [**out**] Variable to store layout array, stored as [nodes, components, elements]. The data for node i, component j, element k in the E-vector is given by i*layout[0] + j*layout[1] + k*layout[2]

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionSetELayout**(*CeedElemRestriction* rstr, *CeedInt* layout[3])

Set the E-vector layout of a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **layout** – [**in**] Variable to containing layout array, stored as [nodes, components, elements]. The data for node i, component j, element k in the E-vector is given by i*layout[0] + j*layout[1] + k*layout[2]

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetData**(*CeedElemRestriction* rstr, void *data)

Get the backend data of a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**in**] CeedElemRestriction
- **data** – [**out**] Variable to store data

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionSetData**(*CeedElemRestriction* rstr, void *data)

Set the backend data of a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**inout**] CeedElemRestriction
- **data** – [**in**] Data to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionReference**(*CeedElemRestriction* rstr)

Increment the reference counter for a CeedElemRestriction.

Backend Developer Functions

Parameters

- **rstr** – [**inout**] ElemRestriction to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

int **CeedElemRestrictionGetFlopsEstimate**(*CeedElemRestriction* rstr, *CeedTransposeMode* t_mode, CeedSize *flops)

Estimate number of FLOPs required to apply CeedElemRestriction in t_mode.

Backend Developer Functions

Parameters

- **rstr** – [**in**] ElemRestriction to estimate FLOPs for
- **t_mode** – [**in**] Apply restriction or transpose
- **flops** – [**out**] Address of variable to hold FLOPs estimate

6.2.4 CeedBasis

int **CeedBasisGetCollocatedGrad**(*CeedBasis* basis, *CeedScalar* *collo_grad_1d)

Return collocated grad matrix.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **collo_grad_1d** – [**out**] Row-major (Q_1d * Q_1d) matrix expressing derivatives of basis functions at quadrature points

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisIsTensor**(*CeedBasis* basis, bool *is_tensor)

Get tensor status for given CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **is_tensor** – [**out**] Variable to store tensor status

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetData**(*CeedBasis* basis, void *data)

Get backend data of a CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **data** – [**out**] Variable to store data

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisSetData**(*CeedBasis* basis, void *data)

Set backend data of a CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**inout**] CeedBasis
- **data** – [**in**] Data to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisReference**(*CeedBasis* basis)

Increment the reference counter for a CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**inout**] Basis to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetNumQuadratureComponents**(*CeedBasis* basis, *CeedEvalMode* eval_mode, *CeedInt* *q_comp)

Get number of Q-vector components for given CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **eval_mode** – [**in**] *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_DIV* to use divergence, *CEED_EVAL_CURL* to use curl.
- **q_comp** – [**out**] Variable to store number of Q-vector components of basis

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetFlopsEstimate**(*CeedBasis* basis, *CeedTransposeMode* t_mode, *CeedEvalMode* eval_mode, CeedSize *flops)

Estimate number of FLOPs required to apply CeedBasis in t_mode and eval_mode.

Backend Developer Functions

Parameters

- **basis** – [**in**] Basis to estimate FLOPs for
- **t_mode** – [**in**] Apply basis or transpose
- **eval_mode** – [**in**] Basis evaluation mode
- **flops** – [**out**] Address of variable to hold FLOPs estimate

int **CeedBasisGetFESpace**(*CeedBasis* basis, CeedFESpace *fe_space)

Get CeedFESpace for a CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **fe_space** – [**out**] Variable to store CeedFESpace

Returns
An error code: 0 - success, otherwise - failure

int **CeedBasisGetTopologyDimension**(*CeedElemTopology* topo, *CeedInt* *dim)

Get dimension for given CeedElemTopology.

Backend Developer Functions

Parameters

- **topo** – [**in**] CeedElemTopology
- **dim** – [**out**] Variable to store dimension of topology

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisGetTensorContract**(*CeedBasis* basis, CeedTensorContract *contract)

Get CeedTensorContract of a CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**in**] CeedBasis
- **contract** – [**out**] Variable to store CeedTensorContract

Returns

An error code: 0 - success, otherwise - failure

int **CeedBasisSetTensorContract**(*CeedBasis* basis, CeedTensorContract contract)

Set CeedTensorContract of a CeedBasis.

Backend Developer Functions

Parameters

- **basis** – [**inout**] CeedBasis
- **contract** – [**in**] CeedTensorContract to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedMatrixMatrixMultiply**(*Ceed* ceed, const *CeedScalar* *mat_A, const *CeedScalar* *mat_B, *CeedScalar* *mat_C, *CeedInt* m, *CeedInt* n, *CeedInt* kk)

Return a reference implementation of matrix multiplication C = A B.

Note: This is a reference implementation for CPU CeedScalar pointers that is not intended for high performance.

Utility Functions

Parameters

- **ceed** – [**in**] Ceed context for error handling
- **mat_A** – [**in**] Row-major matrix A
- **mat_B** – [**in**] Row-major matrix B
- **mat_C** – [**out**] Row-major output matrix C
- **m** – [**in**] Number of rows of C
- **n** – [**in**] Number of columns of C
- **kk** – [**in**] Number of columns of A/rows of B

Returns

An error code: 0 - success, otherwise - failure

int **CeedQRFactorization**(*Ceed* ceed, *CeedScalar* *mat, *CeedScalar* *tau, *CeedInt* m, *CeedInt* n)

Return QR Factorization of a matrix.

Utility Functions

Parameters

- **ceed** – [**in**] Ceed context for error handling

- **mat** – [**inout**] Row-major matrix to be factorized in place
- **tau** – [**inout**] Vector of length m of scaling factors
- **m** – [**in**] Number of rows
- **n** – [**in**] Number of columns

Returns

An error code: 0 - success, otherwise - failure

int **CeedHouseholderApplyQ**(*CeedScalar* *mat_A, const *CeedScalar* *mat_Q, const *CeedScalar* *tau, *CeedTransposeMode* t_mode, *CeedInt* m, *CeedInt* n, *CeedInt* k, *CeedInt* row, *CeedInt* col)

Apply Householder Q matrix.

Compute mat_A = mat_Q mat_A, where mat_Q is mxm and mat_A is mxn.

Utility Functions

Parameters

- **mat_A** – [**inout**] Matrix to apply Householder Q to, in place
- **mat_Q** – [**in**] Householder Q matrix
- **tau** – [**in**] Householder scaling factors
- **t_mode** – [**in**] Transpose mode for application
- **m** – [**in**] Number of rows in A
- **n** – [**in**] Number of columns in A
- **k** – [**in**] Number of elementary reflectors in Q, k<m
- **row** – [**in**] Row stride in A
- **col** – [**in**] Col stride in A

Returns

An error code: 0 - success, otherwise - failure

int **CeedSymmetricSchurDecomposition**(*Ceed* ceed, *CeedScalar* *mat, *CeedScalar* *lambda, *CeedInt* n)

Return symmetric Schur decomposition of the symmetric matrix mat via symmetric QR factorization.

Utility Functions

Parameters

- **ceed** – [**in**] Ceed context for error handling
- **mat** – [**inout**] Row-major matrix to be factorized in place
- **lambda** – [**out**] Vector of length n of eigenvalues
- **n** – [**in**] Number of rows/columns

Returns

An error code: 0 - success, otherwise - failure

int **CeedSimultaneousDiagonalization**(*Ceed* ceed, *CeedScalar* *mat_A, *CeedScalar* *mat_B, *CeedScalar* *mat_X, *CeedScalar* *lambda, *CeedInt* n)

Return Simultaneous Diagonalization of two matrices.

This solves the generalized eigenvalue problem A x = lambda B x, where A and B are symmetric and B is positive definite. We generate the matrix X and vector Lambda such that X^T A X = Lambda and X^T B X = I. This is equivalent to the LAPACK routine 'sygv' with TYPE = 1.

Utility Functions

Parameters

- **`ceed`** – [**in**] Ceed context for error handling
- **`mat_A`** – [**in**] Row-major matrix to be factorized with eigenvalues
- **`mat_B`** – [**in**] Row-major matrix to be factorized to identity
- **`mat_X`** – [**out**] Row-major orthogonal matrix
- **`lambda`** – [**out**] Vector of length n of generalized eigenvalues
- **`n`** – [**in**] Number of rows/columns

Returns

An error code: 0 - success, otherwise - failure

int **`CeedTensorContractCreate`**(*Ceed* ceed, CeedTensorContract *contract)

Create a CeedTensorContract object for a CeedBasis.

Backend Developer Functions

Parameters

- **`ceed`** – [**in**] Ceed object where the CeedTensorContract will be created
- **`contract`** – [**out**] Address of the variable where the newly created CeedTensorContract will be stored.

Returns

An error code: 0 - success, otherwise - failure

int **`CeedTensorContractApply`**(CeedTensorContract contract, *CeedInt* A, *CeedInt* B, *CeedInt* C, *CeedInt* J, const *CeedScalar* *restrict t, *CeedTransposeMode* t_mode, const *CeedInt* add, const *CeedScalar* *restrict u, *CeedScalar* *restrict v)

Apply tensor contraction.

Contracts on the middle index NOTRANSPOSE: v_ajc = t_jb u_abc TRANSPOSE: v_ajc = t_bj u_abc If add != 0, "=" is replaced by "+="

Backend Developer Functions

Parameters

- **`contract`** – [**in**] CeedTensorContract to use
- **`A`** – [**in**] First index of u, v
- **`B`** – [**in**] Middle index of u, one index of t
- **`C`** – [**in**] Last index of u, v
- **`J`** – [**in**] Middle index of v, one index of t
- **`t`** – [**in**] Tensor array to contract against
- **`t_mode`** – [**in**] Transpose mode for t, *CEED_NOTRANSPOSE* for t_jb *CEED_TRANSPOSE* for t_bj
- **`add`** – [**in**] Add mode

- **u** – [in] Input array
- **v** – [out] Output array

Returns

An error code: 0 - success, otherwise - failure

int **CeedTensorContractStridedApply**(CeedTensorContract contract, *CeedInt* A, *CeedInt* B, *CeedInt* C, *CeedInt* D, *CeedInt* J, const *CeedScalar* *restrict t, *CeedTransposeMode* t_mode, const *CeedInt* add, const *CeedScalar* *restrict u, *CeedScalar* *restrict v)

Apply tensor contraction.

Contracts on the middle index NOTRANSPOSE: v_dajc = t_djb u_abc TRANSPOSE: v_ajc = t_dbj u_dabc If add != 0, "=" is replaced by "+="

Backend Developer Functions

Parameters

- **contract** – [in] CeedTensorContract to use
- **A** – [in] First index of u, second index of v
- **B** – [in] Middle index of u, one of last two indices of t
- **C** – [in] Last index of u, v
- **D** – [in] First index of v, first index of t
- **J** – [in] Third index of v, one of last two indices of t
- **t** – [in] Tensor array to contract against
- **t_mode** – [in] Transpose mode for t, *CEED_NOTRANSPOSE* for t_djb *CEED_TRANSPOSE* for t_dbj
- **add** – [in] Add mode
- **u** – [in] Input array
- **v** – [out] Output array

Returns

An error code: 0 - success, otherwise - failure

int **CeedTensorContractGetCeed**(CeedTensorContract contract, *Ceed* *ceed)

Get Ceed associated with a CeedTensorContract.

Backend Developer Functions

Parameters

- **contract** – [in] CeedTensorContract
- **ceed** – [out] Variable to store Ceed

Returns

An error code: 0 - success, otherwise - failure

int **CeedTensorContractGetData**(CeedTensorContract contract, void *data)

Get backend data of a CeedTensorContract.

Backend Developer Functions

Parameters

- **`contract`** – [**in**] CeedTensorContract
- **`data`** – [**out**] Variable to store data

Returns
An error code: 0 - success, otherwise - failure

int **`CeedTensorContractSetData`**(CeedTensorContract contract, void *data)

Set backend data of a CeedTensorContract.

Backend Developer Functions

Parameters

- **`contract`** – [**inout**] CeedTensorContract
- **`data`** – [**in**] Data to set

Returns
An error code: 0 - success, otherwise - failure

int **`CeedTensorContractReference`**(CeedTensorContract contract)

Increment the reference counter for a CeedTensorContract.

Backend Developer Functions

Parameters

- **`contract`** – [**inout**] CeedTensorContract to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

int **`CeedTensorContractReferenceCopy`**(CeedTensorContract tensor, CeedTensorContract *tensor_copy)

Copy the pointer to a CeedTensorContract.

Both pointers should be destroyed with *`CeedTensorContractDestroy()`*.

Note: If the value of `tensor_copy` passed to this function is non-NULL, then it is assumed that `tensor_copy` is a pointer to a CeedTensorContract. This CeedTensorContract will be destroyed if `tensor_copy` is the only reference to this CeedVector.

User Functions

Parameters

- **`tensor`** – [**in**] CeedTensorContract to copy reference to
- **`tensor_copy`** – [**inout**] Variable to store copied reference

Returns
An error code: 0 - success, otherwise - failure

int **`CeedTensorContractDestroy`**(CeedTensorContract *contract)

Destroy a CeedTensorContract.

Backend Developer Functions

Parameters

- **`contract`** – [**inout**] CeedTensorContract to destroy

Returns
An error code: 0 - success, otherwise - failure

6.2.5 CeedQFunction

typedef struct CeedQFunctionField_private ***CeedQFunctionField**

Handle for object describing CeedQFunction fields.

int **CeedQFunctionGetVectorLength**(*CeedQFunction* qf, *CeedInt* *vec_length)

Get the vector length of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **vec_length** – [**out**] Variable to store vector length

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetNumArgs**(*CeedQFunction* qf, *CeedInt* *num_input, *CeedInt* *num_output)

Get the number of inputs and outputs to a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **num_input** – [**out**] Variable to store number of input fields
- **num_output** – [**out**] Variable to store number of output fields

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetKernelName**(*CeedQFunction* qf, char **kernel_name)

Get the name of the user function for a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **kernel_name** – [**out**] Variable to store source path string

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetSourcePath**(*CeedQFunction* qf, char **source_path)

Get the source path string for a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **source_path** – [**out**] Variable to store source path string

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionLoadSourceToBuffer**(*CeedQFunction* qf, char **source_buffer)

Initialize and load QFunction source file into string buffer, including full text of local files in place of `#include "local.h"`.

The `buffer` is set to `NULL` if there is no QFunction source file.

Note: Caller is responsible for freeing the string buffer with `CeedFree()`.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **source_buffer** – [**out**] String buffer for source file contents

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetUserFunction**(*CeedQFunction* qf, CeedQFunctionUser *f)

Get the User Function for a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **f** – [**out**] Variable to store user function

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetContext**(*CeedQFunction* qf, *CeedQFunctionContext* *ctx)

Get global context for a CeedQFunction.

Note: For QFunctions from the Fortran interface, this function will return the Fortran context CeedQFunctionContext.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **ctx** – [**out**] Variable to store CeedQFunctionContext

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetContextData**(*CeedQFunction* qf, *CeedMemType* mem_type, void *data)

Get context data of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **mem_type** – [**in**] Memory type on which to access the data. If the backend uses a different memory type, this will perform a copy.
- **data** – [**out**] Data on memory type mem_type

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionRestoreContextData**(*CeedQFunction* qf, void *data)

Restore context data of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **data** – [**inout**] Data to restore

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetInnerContext**(*CeedQFunction* qf, *CeedQFunctionContext* *ctx)

Get true user context for a CeedQFunction.

Note: For all QFunctions this function will return the user CeedQFunctionContext and not interface context CeedQFunctionContext, if any such object exists.

Parameters

- **qf** – [**in**] CeedQFunction
- **ctx** – [**out**] Variable to store CeedQFunctionContext

Returns

An error code: 0 - success, otherwise - failure Backend Developer Functions

int **CeedQFunctionGetInnerContextData**(*CeedQFunction* qf, *CeedMemType* mem_type, void *data)

Get inner context data of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **mem_type** – [**in**] Memory type on which to access the data. If the backend uses a different memory type, this will perform a copy.
- **data** – [**out**] Data on memory type mem_type

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionRestoreInnerContextData**(*CeedQFunction* qf, void *data)

Restore inner context data of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **data** – [**inout**] Data to restore

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionIsIdentity**(*CeedQFunction* qf, bool *is_identity)

Determine if QFunction is identity.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **is_identity** – [**out**] Variable to store identity status

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionIsContextWritable**(*CeedQFunction* qf, bool *is_writable)

Determine if QFunctionContext is writable.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **is_writable** – [**out**] Variable to store context writeable status

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetData**(*CeedQFunction* qf, void *data)

Get backend data of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] CeedQFunction
- **data** – [**out**] Variable to store data

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionSetData**(*CeedQFunction* qf, void *data)

Set backend data of a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**inout**] CeedQFunction
- **data** – [**in**] Data to set

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionReference**(*CeedQFunction* qf)

Increment the reference counter for a CeedQFunction.

Backend Developer Functions

Parameters

- **qf** – [**inout**] CeedQFunction to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionGetFlopsEstimate**(*CeedQFunction* qf, CeedSize *flops)

Estimate number of FLOPs per quadrature required to apply QFunction.

Backend Developer Functions

Parameters

- **qf** – [**in**] QFunction to estimate FLOPs for
- **flops** – [**out**] Address of variable to hold FLOPs estimate

int **CeedQFunctionContextGetCeed**(*CeedQFunctionContext* ctx, *Ceed* *ceed)

Get the Ceed associated with a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **ceed** – [**out**] Variable to store Ceed

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextHasValidData**(*CeedQFunctionContext* ctx, bool *has_valid_data)

Check for valid data in a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext to check validity
- **has_valid_data** – [**out**] Variable to store validity

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextHasBorrowedDataOfType**(*CeedQFunctionContext* ctx, *CeedMemType* mem_type, bool *has_borrowed_data_of_type)

Check for borrowed data of a specific CeedMemType in a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext to check
- **mem_type** – [**in**] Memory type to check
- **has_borrowed_data_of_type** – [**out**] Variable to store result

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetState**(*CeedQFunctionContext* ctx, uint64_t *state)

Get the state of a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext to retrieve state
- **state** – [**out**] Variable to store state

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetBackendData**(*CeedQFunctionContext* ctx, void *data)

Get backend data of a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **data** – [**out**] Variable to store data

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextSetBackendData**(*CeedQFunctionContext* ctx, void *data)

Set backend data of a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **data** – [**in**] Data to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetFieldLabel**(*CeedQFunctionContext* ctx, const char *field_name, *CeedContextFieldLabel* *field_label)

Get label for a registered QFunctionContext field, or `NULL` if no field has been registered with this `field_name`

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_name** – [**in**] Name of field to retrieve label
- **field_label** – [**out**] Variable to field label

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextSetGeneric**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, CeedContextFieldType field_type, void *values)

Set QFunctionContext field.

Backend Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **field_label** – [**in**] Label of field to set
- **field_type** – [**in**] Type of field to set
- **values** – [**in**] Value to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetGenericRead**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, CeedContextFieldType field_type, size_t *num_values, void *values)

Get QFunctionContext field data, read-only.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_label** – [**in**] Label of field to read
- **field_type** – [**in**] Type of field to read
- **num_values** – [**out**] Number of values in the field label
- **values** – [**out**] Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRestoreGenericRead**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, CeedContextFieldType field_type, void *values)

Restore QFunctionContext field data, read-only.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_label** – [**in**] Label of field to restore
- **field_type** – [**in**] Type of field to restore
- **values** – [**out**] Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextSetDouble**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, double *values)

Set QFunctionContext field holding a double precision value.

Backend Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **field_label** – [**in**] Label for field to set
- **values** – [**in**] Values to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetDoubleRead**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, size_t *num_values, const double **values)

Get QFunctionContext field holding a double precision value, read-only.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_label** – [**in**] Label for field to get
- **num_values** – [**out**] Number of values in the field label
- **values** – [**out**] Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRestoreDoubleRead**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, const double **values)

Restore QFunctionContext field holding a double precision value, read-only.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_label** – [**in**] Label for field to restore
- **values** – [**out**] Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextSetInt32**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, int *values)

Set QFunctionContext field holding an int32 value.

Backend Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **field_label** – [**in**] Label for field to set
- **values** – [**in**] Values to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetInt32Read**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, size_t *num_values, const int **values)

Get QFunctionContext field holding a int32 value, read-only.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_label** – [**in**] Label for field to get
- **num_values** – [**out**] Number of values in the field label
- **values** – [**out**] Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRestoreInt32Read**(*CeedQFunctionContext* ctx, *CeedContextFieldLabel* field_label, const int **values)

Restore QFunctionContext field holding a int32 value, read-only.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_label** – [**in**] Label for field to restore
- **values** – [**out**] Pointer to context values

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetDataDestroy**(*CeedQFunctionContext* ctx, *CeedMemType* *f_mem_type, CeedQFunctionContextDataDestroyUser *f)

Get additional destroy routine for CeedQFunctionContext user data.

Backend Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext to get user destroy function
- **f_mem_type** – [**out**] Memory type to use when passing data into `f`
- **f** – [**out**] Additional routine to use to destroy user data

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextReference**(*CeedQFunctionContext* ctx)

Increment the reference counter for a CeedQFunctionContext.

Backend Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext to increment the reference counter

Returns

An error code: 0 - success, otherwise - failure

6.2.5.1 Macros

CEED_QFUNCTION_ATTR

This macro defines compiler attributes to the CEED_QFUNCTION to force inlining for called functions.

The `inline` declaration does not necessarily enforce a compiler to inline a function. This can be detrimental to performance, so here we force inlining to occur unless inlining has been forced off (like during debugging).

6.2.6 CeedOperator

typedef struct CeedOperatorField_private ***CeedOperatorField**

Handle for object describing CeedOperator fields.

int **CeedOperatorGetNumArgs**(*CeedOperator* op, *CeedInt* *num_args)

Get the number of arguments associated with a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [in] CeedOperator
- **num_args** – [out] Variable to store vector number of arguments

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorIsSetupDone**(*CeedOperator* op, bool *is_setup_done)

Get the setup status of a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [in] CeedOperator
- **is_setup_done** – [out] Variable to store setup status

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetQFunction**(*CeedOperator* op, *CeedQFunction* *qf)

Get the QFunction associated with a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [in] CeedOperator
- **qf** – [out] Variable to store QFunction

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorIsComposite**(*CeedOperator* op, bool *is_composite)

Get a boolean value indicating if the CeedOperator is composite.

Backend Developer Functions

Parameters

- **op** – [in] CeedOperator
- **is_composite** – [out] Variable to store composite status

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetData**(*CeedOperator* op, void *data)

Get the backend data of a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [**in**] CeedOperator
- **data** – [**out**] Variable to store data

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorSetData**(*CeedOperator* op, void *data)

Set the backend data of a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [**inout**] CeedOperator
- **data** – [**in**] Data to set

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorReference**(*CeedOperator* op)

Increment the reference counter for a CeedOperator.

Backend Developer Functions

Parameters

- **op** – [**inout**] CeedOperator to increment the reference counter

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorSetSetupDone**(*CeedOperator* op)

Set the setup flag of a CeedOperator to True.

Backend Developer Functions

Parameters

- **op** – [**inout**] CeedOperator

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorCreateActivePointBlockRestriction**(*CeedElemRestriction* rstr, *CeedElemRestriction* *point_block_rstr)

Create point block restriction for active operator field.

Backend Developer Functions

Parameters

- **rstr** – [**in**] Original CeedElemRestriction for active field
- **point_block_rstr** – [**out**] Address of the variable where the newly created CeedElemRestriction will be stored

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataCreate**(*Ceed* ceed, CeedQFunctionAssemblyData *data)

Create object holding CeedQFunction assembly data for CeedOperator.

Backend Developer Functions

Parameters

- **ceed** – [**in**] A Ceed object where the CeedQFunctionAssemblyData will be created
- **data** – [**out**] Address of the variable where the newly created CeedQFunctionAssemblyData will be stored

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataReference**(CeedQFunctionAssemblyData data)

Increment the reference counter for a CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **data** – [**inout**] CeedQFunctionAssemblyData to increment the reference counter

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataSetReuse**(CeedQFunctionAssemblyData data, bool reuse_data)

Set re-use of CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **data** – [**inout**] CeedQFunctionAssemblyData to mark for reuse
- **reuse_data** – [**in**] Boolean flag indicating data re-use

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataSetUpdateNeeded**(CeedQFunctionAssemblyData data, bool needs_data_update)

Mark QFunctionAssemblyData as stale.

Backend Developer Functions

Parameters

- **data** – [**inout**] CeedQFunctionAssemblyData to mark as stale
- **needs_data_update** – [**in**] Boolean flag indicating if update is needed or completed

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataIsUpdateNeeded**(CeedQFunctionAssemblyData data, bool *is_update_needed)

Determine if QFunctionAssemblyData needs update.

Backend Developer Functions

Parameters

- **`data`** – [**in**] CeedQFunctionAssemblyData to mark as stale
- **`is_update_needed`** – [**out**] Boolean flag indicating if re-assembly is required

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionAssemblyDataReferenceCopy`**(CeedQFunctionAssemblyData data, CeedQFunctionAssemblyData *data_copy)

Copy the pointer to a CeedQFunctionAssemblyData.

Both pointers should be destroyed with `CeedCeedQFunctionAssemblyDataDestroy()`.

Note: If the value of `data_copy` passed to this function is non-NULL, then it is assumed that `*data_copy` is a pointer to a CeedQFunctionAssemblyData. This CeedQFunctionAssemblyData will be destroyed if `data_copy` is the only reference to this CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **`data`** – [**in**] CeedQFunctionAssemblyData to copy reference to
- **`data_copy`** – [**inout**] Variable to store copied reference

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionAssemblyDataIsSetup`**(CeedQFunctionAssemblyData data, bool *is_setup)

Get setup status for internal objects for CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **`data`** – [**in**] CeedQFunctionAssemblyData to retrieve status
- **`is_setup`** – [**out**] Boolean flag for setup status

Returns

An error code: 0 - success, otherwise - failure

int **`CeedQFunctionAssemblyDataSetObjects`**(CeedQFunctionAssemblyData data, *CeedVector* vec, *CeedElemRestriction* rstr)

Set internal objects for CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **`data`** – [**inout**] CeedQFunctionAssemblyData to set objects
- **`vec`** – [**in**] CeedVector to store assembled CeedQFunction at quadrature points
- **`rstr`** – [**in**] CeedElemRestriction for CeedVector containing assembled CeedQFunction

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataGetObjects**(CeedQFunctionAssemblyData data, *CeedVector* *vec, *CeedElemRestriction* *rstr)

Get internal objects for CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **data** – [**inout**] CeedQFunctionAssemblyData to set objects
- **vec** – [**out**] CeedVector to store assembled CeedQFunction at quadrature points
- **rstr** – [**out**] CeedElemRestriction for CeedVector containing assembled CeedQFunction

Returns

An error code: 0 - success, otherwise - failure

int **CeedQFunctionAssemblyDataDestroy**(CeedQFunctionAssemblyData *data)

Destroy CeedQFunctionAssemblyData.

Backend Developer Functions

Parameters

- **data** – [**inout**] CeedQFunctionAssemblyData to destroy

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetOperatorAssemblyData**(*CeedOperator* op, CeedOperatorAssemblyData *data)

Get CeedOperatorAssemblyData.

Backend Developer Functions

Parameters

- **op** – [**in**] CeedOperator to assemble
- **data** – [**out**] CeedQFunctionAssemblyData

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorAssemblyDataCreate**(*Ceed* ceed, *CeedOperator* op, CeedOperatorAssemblyData *data)

Create object holding CeedOperator assembly data.

The CeedOperatorAssemblyData holds an array with references to every active CeedBasis used in the CeedOperator. An array with references to the corresponding active CeedElemRestrictions is also stored. For each active CeedBasis, the CeedOperatorAssemblyData holds an array of all input and output CeedEvalModes for this CeedBasis. The CeedOperatorAssemblyData holds an array of offsets for indexing into the assembled CeedQFunction arrays to the row representing each CeedEvalMode. The number of input columns across all active bases for the assembled CeedQFunction is also stored. Lastly, the CeedOperatorAssembly data holds assembled matrices representing the full action of the CeedBasis for all CeedEvalModes.

Backend Developer Functions

Parameters

- **ceed** – [**in**] Ceed object where the CeedOperatorAssemblyData will be created
- **op** – [**in**] CeedOperator to be assembled

- **data** – [**out**] Address of the variable where the newly created CeedOperatorAssemblyData will be stored

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorAssemblyDataGetEvalModes**(CeedOperatorAssemblyData data, *CeedInt* *num_active_bases_in, *CeedInt* **num_eval_modes_in, const *CeedEvalMode* ***eval_modes_in, CeedSize ***eval_mode_offsets_in, *CeedInt* *num_active_bases_out, *CeedInt* **num_eval_modes_out, const *CeedEvalMode* ***eval_modes_out, CeedSize ***eval_mode_offsets_out, CeedSize *num_output_components)

Get CeedOperator CeedEvalModes for assembly.

Note: See CeedOperatorAssemblyDataCreate for a full description of the data stored in this object.

Backend Developer Functions

Parameters

- **data** – [**in**] CeedOperatorAssemblyData
- **num_active_bases_in** – [**out**] Total number of active bases for input
- **num_eval_modes_in** – [**out**] Pointer to hold array of numbers of input CeedEvalModes, or NULL. `eval_modes_in[0]` holds an array of eval modes for the first active basis.
- **eval_modes_in** – [**out**] Pointer to hold arrays of input CeedEvalModes, or NULL.
- **eval_mode_offsets_in** – [**out**] Pointer to hold arrays of input offsets at each quadrature point.
- **num_active_bases_out** – [**out**] Total number of active bases for output
- **num_eval_modes_out** – [**out**] Pointer to hold array of numbers of output CeedEvalModes, or NULL
- **eval_modes_out** – [**out**] Pointer to hold arrays of output CeedEvalModes, or NULL.
- **eval_mode_offsets_out** – [**out**] Pointer to hold arrays of output offsets at each quadrature point
- **num_output_components** – [**out**] The number of columns in the assembled CeedQFunction matrix for each quadrature point, including contributions of all active bases

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorAssemblyDataGetBases**(CeedOperatorAssemblyData data, *CeedInt* *num_active_bases_in, *CeedBasis* **active_bases_in, const *CeedScalar* ***assembled_bases_in, *CeedInt* *num_active_bases_out, *CeedBasis* **active_bases_out, const *CeedScalar* ***assembled_bases_out)

Get CeedOperator CeedBasis data for assembly.

Note: See CeedOperatorAssemblyDataCreate for a full description of the data stored in this object.

Backend Developer Functions

Parameters

- **`data`** – [**in**] CeedOperatorAssemblyData
- **`num_active_bases_in`** – [**out**] Number of active input bases, or NULL
- **`active_bases_in`** – [**out**] Pointer to hold active input CeedBasis, or NULL
- **`assembled_bases_in`** – [**out**] Pointer to hold assembled active input B, or NULL
- **`num_active_bases_out`** – [**out**] Number of active output bases, or NULL
- **`active_bases_out`** – [**out**] Pointer to hold active output CeedBasis, or NULL
- **`assembled_bases_out`** – [**out**] Pointer to hold assembled active output B, or NULL

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorAssemblyDataGetElemRestrictions`**(CeedOperatorAssemblyData data, *CeedInt* *num_active_elem_rstrs_in, *CeedElemRestriction* **active_elem_rstrs_in, *CeedInt* *num_active_elem_rstrs_out, *CeedElemRestriction* **active_elem_rstrs_out)

Get CeedOperator CeedBasis data for assembly.

Note: See CeedOperatorAssemblyDataCreate for a full description of the data stored in this object.

Backend Developer Functions

Parameters

- **`data`** – [**in**] CeedOperatorAssemblyData
- **`num_active_elem_rstrs_in`** – [**out**] Number of active input element restrictions, or NULL
- **`active_elem_rstrs_in`** – [**out**] Pointer to hold active input CeedElemRestrictions, or NULL
- **`num_active_elem_rstrs_out`** – [**out**] Number of active output element restrictions, or NULL
- **`active_elem_rstrs_out`** – [**out**] Pointer to hold active output CeedElemRestrictions, or NULL

Returns

An error code: 0 - success, otherwise - failure

int **`CeedOperatorAssemblyDataDestroy`**(CeedOperatorAssemblyData *data)

Destroy CeedOperatorAssemblyData.

Backend Developer Functions

Parameters

- **`data`** – [**inout**] CeedOperatorAssemblyData to destroy

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetFallback**(*CeedOperator* op, *CeedOperator* *op_fallback)

Retrieve fallback CeedOperator with a reference Ceed for advanced CeedOperator functionality.

Backend Developer Functions

Parameters

- **op** – [**in**] CeedOperator to retrieve fallback for
- **op_fallback** – [**out**] Fallback CeedOperator

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetFallbackParent**(*CeedOperator* op, *CeedOperator* *parent)

Get the parent CeedOperator for a fallback CeedOperator.

Backend Developer Functions

Parameters

- **op** – [**in**] CeedOperator context
- **parent** – [**out**] Variable to store parent CeedOperator context

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorGetFallbackParentCeed**(*CeedOperator* op, *Ceed* *parent)

Get the Ceed context of the parent CeedOperator for a fallback CeedOperator.

Backend Developer Functions

Parameters

- **op** – [**in**] CeedOperator context
- **parent** – [**out**] Variable to store parent Ceed context

Returns

An error code: 0 - success, otherwise - failure

6.3 Internal Functions

These functions are intended to be used by library developers of libCEED and can generally be found in *ceed-impl.h*.

6.3.1 Ceed

int **CeedRegisterImpl**(const char *prefix, int (*init)(const char*, *Ceed*), unsigned int priority)

Register a Ceed backend internally.

Note: Backends should call `CeedRegister` instead.

Library Developer Functions

Parameters

- **prefix** – [**in**] Prefix of resources for this backend to respond to. For example, the reference backend responds to “/cpu/self”.

- **init** – [**in**] Initialization function called by *CeedInit()* when the backend is selected to drive the requested resource.
- **priority** – [**in**] Integer priority. Lower values are preferred in case the resource requested by *CeedInit()* has non-unique best prefix match.

Returns

An error code: 0 - success, otherwise - failure

6.3.2 CeedVector

6.3.3 CeedElemRestriction

int **CeedPermutePadOffsets**(const *CeedInt* *offsets, *CeedInt* *block_offsets, *CeedInt* num_block, *CeedInt* num_elem, *CeedInt* block_size, *CeedInt* elem_size)

Permute and pad offsets for a blocked restriction.

Utility Functions

Parameters

- **offsets** – [**in**] Array of shape [*num_elem*, *elem_size*].
- **block_offsets** – [**out**] Array of permuted and padded array values of shape [*num_block*, *elem_size*, *block_size*].
- **num_block** – [**in**] Number of blocks
- **num_elem** – [**in**] Number of elements
- **block_size** – [**in**] Number of elements in a block
- **elem_size** – [**in**] Size of each element

Returns

An error code: 0 - success, otherwise - failure

int **CeedPermutePadOrients**(const bool *orients, bool *block_orients, *CeedInt* num_block, *CeedInt* num_elem, *CeedInt* block_size, *CeedInt* elem_size)

Permute and pad orientations for a blocked restriction.

Utility Functions

Parameters

- **orients** – [**in**] Array of shape [*num_elem*, *elem_size*].
- **block_orients** – [**out**] Array of permuted and padded array values of shape [*num_block*, *elem_size*, *block_size*].
- **num_block** – [**in**] Number of blocks
- **num_elem** – [**in**] Number of elements
- **block_size** – [**in**] Number of elements in a block
- **elem_size** – [**in**] Size of each element

Returns

An error code: 0 - success, otherwise - failure

int **CeedPermutePadCurlOrients**(const CeedInt8 *curl_orients, CeedInt8 *block_curl_orients, *CeedInt* num_block, *CeedInt* num_elem, *CeedInt* block_size, *CeedInt* elem_size)

Permute and pad curl-conforming orientations for a blocked restriction.

Utility Functions

Parameters

- **curl_orients** – [**in**] Array of shape [*num_elem*, *3* * elem_size].
- **block_curl_orients** – [**out**] Array of permuted and padded array values of shape [*num_block*, *elem_size*, *block_size*].
- **num_block** – [**in**] Number of blocks
- **num_elem** – [**in**] Number of elements
- **block_size** – [**in**] Number of elements in a block
- **elem_size** – [**in**] Size of each element

Returns

An error code: 0 - success, otherwise - failure

6.3.4 CeedBasis

static int **CeedChebyshevPolynomialsAtPoint**(*CeedScalar* x, *CeedInt* n, *CeedScalar* *chebyshev_x)

Compute Chebyshev polynomial values at a point.

Library Developer Functions

Parameters

- **x** – [**in**] Coordinate to evaluate Chebyshev polynomials at
- **n** – [**in**] Number of Chebyshev polynomials to evaluate, n >= 2
- **chebyshev_x** – [**out**] Array of Chebyshev polynomial values

Returns

An error code: 0 - success, otherwise - failure

static int **CeedChebyshevDerivativeAtPoint**(*CeedScalar* x, *CeedInt* n, *CeedScalar* *chebyshev_dx)

Compute values of the derivative of Chebyshev polynomials at a point.

Library Developer Functions

Parameters

- **x** – [**in**] Coordinate to evaluate derivative of Chebyshev polynomials at
- **n** – [**in**] Number of Chebyshev polynomials to evaluate, n >= 2
- **chebyshev_dx** – [**out**] Array of Chebyshev polynomial derivative values

Returns

An error code: 0 - success, otherwise - failure

static int **CeedHouseholderReflect**(*CeedScalar* *A, const *CeedScalar* *v, *CeedScalar* b, *CeedInt* m, *CeedInt* n, *CeedInt* row, *CeedInt* col)

Compute Householder reflection.

Computes A = (I - b v v^T) A, where A is an mxn matrix indexed as A[i*row + j*col]

Library Developer Functions

Parameters

- **A** – [**inout**] Matrix to apply Householder reflection to, in place
- **v** – [**in**] Householder vector
- **b** – [**in**] Scaling factor
- **m** – [**in**] Number of rows in A
- **n** – [**in**] Number of columns in A
- **row** – [**in**] Row stride
- **col** – [**in**] Col stride

Returns

An error code: 0 - success, otherwise - failure

static int **CeedGivensRotation**(*CeedScalar* *A, *CeedScalar* c, *CeedScalar* s, *CeedTransposeMode* t_mode, *CeedInt* i, *CeedInt* k, *CeedInt* m, *CeedInt* n)

Compute Givens rotation.

Computes A = G A (or G^T A in transpose mode), where A is an mxn matrix indexed as A[i*n + j*m]

Library Developer Functions

Parameters

- **A** – [**inout**] Row major matrix to apply Givens rotation to, in place
- **c** – [**in**] Cosine factor
- **s** – [**in**] Sine factor
- **t_mode** – [**in**] *CEED_NOTRANSPOSE* to rotate the basis counter-clockwise, which has the effect of rotating columns of A clockwise; *CEED_TRANSPOSE* for the opposite rotation
- **i** – [**in**] First row/column to apply rotation
- **k** – [**in**] Second row/column to apply rotation
- **m** – [**in**] Number of rows in A
- **n** – [**in**] Number of columns in A

Returns

An error code: 0 - success, otherwise - failure

static int **CeedScalarView**(const char *name, const char *fp_fmt, *CeedInt* m, *CeedInt* n, const *CeedScalar* *a, FILE *stream)

View an array stored in a CeedBasis.

Library Developer Functions

Parameters

- **name** – [**in**] Name of array
- **fp_fmt** – [**in**] Printing format

- **m** – [in] Number of rows in array
- **n** – [in] Number of columns in array
- **a** – [in] Array to be viewed
- **stream** – [in] Stream to view to, e.g., stdout

Returns
An error code: 0 - success, otherwise - failure

static int **CeedBasisCreateProjectionMatrices**(*CeedBasis* basis_from, *CeedBasis* basis_to, *CeedScalar* **interp_project, *CeedScalar* **grad_project)

Create the interpolation and gradient matrices for projection from the nodes of `basis_from` to the nodes of `basis_to`.

The interpolation is given by `interp_project = interp_to^+ * interp_from`, where the pseudoinverse `interp_to^+` is given by QR factorization. The gradient is given by `grad_project = interp_to^+ * grad_from`, and is only computed for H^1 spaces otherwise it should not be used.

Note: `basis_from` and `basis_to` must have compatible quadrature spaces.

Library Developer Functions

Parameters

- **basis_from** – [in] CeedBasis to project from
- **basis_to** – [in] CeedBasis to project to
- **interp_project** – [out] Address of the variable where the newly created interpolation matrix will be stored.
- **grad_project** – [out] Address of the variable where the newly created gradient matrix will be stored.

Returns
An error code: 0 - success, otherwise - failure

6.3.5 CeedQFunction

int **CeedQFunctionRegister**(const char *name, const char *source, *CeedInt* vec_length, CeedQFunctionUser f, int (*init)(*Ceed*, const char*, *CeedQFunction*))

Register a gallery QFunction.

Library Developer Functions

Parameters

- **name** – [in] Name for this backend to respond to
- **source** – [in] Absolute path to source of QFunction, “\path\CEED_DIR\gallery\folder\file.h:function_name”
- **vec_length** – [in] Vector length. Caller must ensure that number of quadrature points is a multiple of vec_length.
- **f** – [in] Function pointer to evaluate action at quadrature points. See Public API for CeedQFunction.
- **init** – [in] Initialization function called by CeedQFunctionInit() when the QFunction is selected.

Returns
An error code: 0 - success, otherwise - failure

static int **`CeedQFunctionFieldSet`**(*CeedQFunctionField* *f, const char *field_name, *CeedInt* size, *CeedEvalMode* eval_mode)

Set a CeedQFunction field, used by CeedQFunctionAddInput/Output.

Library Developer Functions

Parameters

- **`f`** – [**out**] CeedQFunctionField
- **`field_name`** – [**in**] Name of QFunction field
- **`size`** – [**in**] Size of QFunction field, (num_comp * 1) for *CEED_EVAL_NONE* and *CEED_EVAL_WEIGHT*, (num_comp * 1) for *CEED_EVAL_INTERP* for an H^1 space or (num_comp * dim) for an H(div) or H(curl) space, (num_comp * dim) for *CEED_EVAL_GRAD*, or (num_comp * 1) for *CEED_EVAL_DIV*, and (num_comp * curl_dim) with curl_dim = 1 if dim < 3 else dim for *CEED_EVAL_CURL*.
- **`eval_mode`** – [**in**] *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_WEIGHT* to use quadrature weights, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_DIV* to use divergence, *CEED_EVAL_CURL* to use curl.

Returns
An error code: 0 - success, otherwise - failure

static int **`CeedQFunctionFieldView`**(*CeedQFunctionField* field, *CeedInt* field_number, bool in, FILE *stream)

View a field of a CeedQFunction.

Utility Functions

Parameters

- **`field`** – [**in**] QFunction field to view
- **`field_number`** – [**in**] Number of field being viewed
- **`in`** – [**in**] true for input field, false for output
- **`stream`** – [**in**] Stream to view to, e.g., stdout

Returns
An error code: 0 - success, otherwise - failure

int **`CeedQFunctionSetFortranStatus`**(*CeedQFunction* qf, bool status)

Set flag to determine if Fortran interface is used.

Backend Developer Functions

Parameters

- **`qf`** – [**inout**] CeedQFunction
- **`status`** – [**in**] Boolean value to set as Fortran status

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextGetFieldIndex**(*CeedQFunctionContext* ctx, const char *field_name, *CeedInt* *field_index)

Get index for QFunctionContext field.

Library Developer Functions

Parameters

- **ctx** – [**in**] CeedQFunctionContext
- **field_name** – [**in**] Name of field
- **field_index** – [**out**] Index of field, or -1 if field is not registered

Returns
An error code: 0 - success, otherwise - failure

int **CeedQFunctionContextRegisterGeneric**(*CeedQFunctionContext* ctx, const char *field_name, size_t field_offset, const char *field_description, CeedContextFieldType field_type, size_t field_size, size_t num_values)

Common function for registering QFunctionContext fields.

Library Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext
- **field_name** – [**in**] Name of field to register
- **field_offset** – [**in**] Offset of field to register
- **field_description** – [**in**] Description of field, or NULL for none
- **field_type** – [**in**] Field data type, such as double or int32
- **field_size** – [**in**] Size of field, in bytes
- **num_values** – [**in**] Number of values to register, must be contiguous in memory

Returns
An error code: 0 - success, otherwise - failure

static int **CeedQFunctionContextDestroyData**(*CeedQFunctionContext* ctx)

Destroy user data held by CeedQFunctionContext, using function set by CeedQFunctionContextSetDataDestroy, if applicable.

Library Developer Functions

Parameters

- **ctx** – [**inout**] CeedQFunctionContext to destroy user data

Returns
An error code: 0 - success, otherwise - failure

6.3.6 CeedOperator

static int **CeedOperatorCheckField**(*Ceed* ceed, *CeedQFunctionField* qf_field, *CeedElemRestriction* r, *CeedBasis* b)

Check if a CeedOperator Field matches the QFunction Field.

Library Developer Functions

Parameters

- **ceed** – [**in**] Ceed object for error handling
- **qf_field** – [**in**] QFunction Field matching Operator Field
- **r** – [**in**] Operator Field ElemRestriction
- **b** – [**in**] Operator Field Basis

Returns

An error code: 0 - success, otherwise - failure

static int **CeedOperatorFieldView**(*CeedOperatorField* field, *CeedQFunctionField* qf_field, *CeedInt* field_number, bool sub, bool input, FILE *stream)

View a field of a CeedOperator.

Utility Functions

Parameters

- **field** – [**in**] Operator field to view
- **qf_field** – [**in**] QFunction field (carries field name)
- **field_number** – [**in**] Number of field being viewed
- **sub** – [**in**] true indicates sub-operator, which increases indentation; false for top-level operator
- **input** – [**in**] true for an input field; false for output field
- **stream** – [**in**] Stream to view to, e.g., stdout

Returns

An error code: 0 - success, otherwise - failure

int **CeedOperatorSingleView**(*CeedOperator* op, bool sub, FILE *stream)

View a single CeedOperator.

Utility Functions

Parameters

- **op** – [**in**] CeedOperator to view
- **sub** – [**in**] Boolean flag for sub-operator
- **stream** – [**in**] Stream to write; typically stdout/stderr or a file

Returns

Error code: 0 - success, otherwise - failure

int **CeedOperatorGetActiveBasis**(*CeedOperator* op, *CeedBasis* *active_basis)

Find the active vector basis for a non-composite CeedOperator.

Library Developer Functions

Parameters

- **`op`** – [**in**] CeedOperator to find active basis for
- **`active_basis`** – [**out**] Basis for active input vector or NULL for composite operator

Returns
An error code: 0 - success, otherwise - failure

int **`CeedOperatorGetActiveBases`**(*CeedOperator* op, *CeedBasis* *active_input_basis, *CeedBasis* *active_output_basis)

Find the active input and output vector bases for a non-composite CeedOperator.

Library Developer Functions

Parameters

- **`op`** – [**in**] CeedOperator to find active bases for
- **`active_input_basis`** – [**out**] Basis for active input vector or NULL for composite operator
- **`active_output_basis`** – [**out**] Basis for active output vector or NULL for composite operator

Returns
An error code: 0 - success, otherwise - failure

int **`CeedOperatorGetActiveElemRestriction`**(*CeedOperator* op, *CeedElemRestriction* *active_rstr)

Find the active vector ElemRestriction for a non-composite CeedOperator.

Utility Functions

Parameters

- **`op`** – [**in**] CeedOperator to find active ElemRestriction for
- **`active_rstr`** – [**out**] ElemRestriction for active input vector or NULL for composite operator

Returns
An error code: 0 - success, otherwise - failure

int **`CeedOperatorGetActiveElemRestrictions`**(*CeedOperator* op, *CeedElemRestriction* *active_input_rstr, *CeedElemRestriction* *active_output_rstr)

Find the active input and output vector ElemRestrictions for a non-composite CeedOperator.

Utility Functions

Parameters

- **`op`** – [**in**] CeedOperator to find active ElemRestrictions for
- **`active_input_rstr`** – [**out**] ElemRestriction for active input vector or NULL for composite operator
- **`active_output_rstr`** – [**out**] ElemRestriction for active output vector or NULL for composite operator

Returns
An error code: 0 - success, otherwise - failure

static int **CeedOperatorContextSetGeneric**(*CeedOperator* op, *CeedContextFieldLabel* field_label, CeedContextFieldType field_type, void *values)

Set QFunctionContext field values of the specified type.

For composite operators, the value is set in all sub-operator QFunctionContexts that have a matching `field_name`. A non-zero error code is returned for single operators that do not have a matching field of the same type or composite operators that do not have any field of a matching type.

User Functions

Parameters

- **op** – [**inout**] CeedOperator
- **field_label** – [**in**] Label of field to set
- **field_type** – [**in**] Type of field to set
- **values** – [**in**] Values to set

Returns

An error code: 0 - success, otherwise - failure

static int **CeedOperatorContextGetGenericRead**(*CeedOperator* op, *CeedContextFieldLabel* field_label, CeedContextFieldType field_type, size_t *num_values, void *values)

Get QFunctionContext field values of the specified type, read-only.

For composite operators, the values retrieved are for the first sub-operator QFunctionContext that have a matching `field_name`. A non-zero error code is returned for single operators that do not have a matching field of the same type or composite operators that do not have any field of a matching type.

User Functions

Parameters

- **op** – [**inout**] CeedOperator
- **field_label** – [**in**] Label of field to set
- **field_type** – [**in**] Type of field to set
- **num_values** – [**out**] Number of values of type `field_type` in array `values`
- **values** – [**out**] Values in the label

Returns

An error code: 0 - success, otherwise - failure

static int **CeedOperatorContextRestoreGenericRead**(*CeedOperator* op, *CeedContextFieldLabel* field_label, CeedContextFieldType field_type, void *values)

Restore QFunctionContext field values of the specified type, read-only.

For composite operators, the values restored are for the first sub-operator QFunctionContext that have a matching `field_name`. A non-zero error code is returned for single operators that do not have a matching field of the same type or composite operators that do not have any field of a matching type.

User Functions

Parameters

- **op** – [**inout**] CeedOperator
- **field_label** – [**in**] Label of field to set

- **field_type** – [**in**] Type of field to set
- **values** – [**in**] Values array to restore

Returns
An error code: 0 - success, otherwise - failure

static int **CeedQFunctionCreateFallback**(*Ceed* fallback_ceed, *CeedQFunction* qf, *CeedQFunction* *qf_fallback)

Duplicate a CeedQFunction with a reference Ceed to fallback for advanced CeedOperator functionality.

Library Developer Functions

Parameters

- **fallback_ceed** – [**in**] Ceed on which to create fallback CeedQFunction
- **qf** – [**in**] CeedQFunction to create fallback for
- **qf_fallback** – [**out**] fallback CeedQFunction

Returns
An error code: 0 - success, otherwise - failure

static int **CeedOperatorCreateFallback**(*CeedOperator* op)

Duplicate a CeedOperator with a reference Ceed to fallback for advanced CeedOperator functionality.

Library Developer Functions

Parameters

- **op** – [**inout**] CeedOperator to create fallback for

Returns
An error code: 0 - success, otherwise - failure

static inline int **CeedOperatorGetBasisPointer**(*CeedBasis* basis, *CeedEvalMode* eval_mode, const *CeedScalar* *identity, const *CeedScalar* **basis_ptr)

Select correct basis matrix pointer based on CeedEvalMode.

Library Developer Functions

Parameters

- **basis** – [**in**] CeedBasis from which to get the basis matrix
- **eval_mode** – [**in**] Current basis evaluation mode
- **identity** – [**in**] Pointer to identity matrix
- **basis_ptr** – [**out**] Basis pointer to set

static inline int **CeedSingleOperatorAssembleAddDiagonal_Core**(*CeedOperator* op, *CeedRequest* *request, const bool is_point_block, *CeedVector* assembled)

Core logic for assembling operator diagonal or point block diagonal.

Library Developer Functions

Parameters

- **op** – [**in**] CeedOperator to assemble point block diagonal

- **request** – [in] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE
- **is_point_block** – [in] Boolean flag to assemble diagonal or point block diagonal
- **assembled** – [out] CeedVector to store assembled diagonal

Returns
An error code: 0 - success, otherwise - failure

static inline int **CeedCompositeOperatorLinearAssembleAddDiagonal**(*CeedOperator* op, *CeedRequest* *request, const bool is_point_block, *CeedVector* assembled)

Core logic for assembling composite operator diagonal.

Library Developer Functions

Parameters

- **op** – [in] CeedOperator to assemble point block diagonal
- **request** – [in] Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE
- **is_point_block** – [in] Boolean flag to assemble diagonal or point block diagonal
- **assembled** – [out] CeedVector to store assembled diagonal

Returns
An error code: 0 - success, otherwise - failure

static int **CeedSingleOperatorAssembleSymbolic**(*CeedOperator* op, *CeedInt* offset, *CeedInt* *rows, *CeedInt* *cols)

Build nonzero pattern for non-composite operator.

Users should generally use *CeedOperatorLinearAssembleSymbolic()*

Library Developer Functions

Parameters

- **op** – [in] CeedOperator to assemble nonzero pattern
- **offset** – [in] Offset for number of entries
- **rows** – [out] Row number for each entry
- **cols** – [out] Column number for each entry

Returns
An error code: 0 - success, otherwise - failure

static int **CeedSingleOperatorAssemble**(*CeedOperator* op, *CeedInt* offset, *CeedVector* values)

Assemble nonzero entries for non-composite operator.

Users should generally use *CeedOperatorLinearAssemble()*

Library Developer Functions

Parameters

- **op** – [in] CeedOperator to assemble
- **offset** – [in] Offset for number of entries
- **values** – [out] Values to assemble into matrix

Returns

An error code: 0 - success, otherwise - failure

static int **CeedSingleOperatorAssemblyCountEntries**(*CeedOperator* op, CeedSize *num_entries)

Count number of entries for assembled CeedOperator.

Utility Functions

Parameters

- **op** – [**in**] CeedOperator to assemble
- **num_entries** – [**out**] Number of entries in assembled representation

Returns

An error code: 0 - success, otherwise - failure

static int **CeedSingleOperatorMultigridLevel**(*CeedOperator* op_fine, *CeedVector* p_mult_fine, *CeedElemRestriction* rstr_coarse, *CeedBasis* basis_coarse, *CeedBasis* basis_c_to_f, *CeedOperator* *op_coarse, *CeedOperator* *op_prolong, *CeedOperator* *op_restrict)

Common code for creating a multigrid coarse operator and level transfer operators for a CeedOperator.

Library Developer Functions

Parameters

- **op_fine** – [**in**] Fine grid operator
- **p_mult_fine** – [**in**] L-vector multiplicity in parallel gather/scatter, or NULL if not creating prolongation/restriction operators
- **rstr_coarse** – [**in**] Coarse grid restriction
- **basis_coarse** – [**in**] Coarse grid active vector basis
- **basis_c_to_f** – [**in**] Basis for coarse to fine interpolation, or NULL if not creating prolongation/restriction operators
- **op_coarse** – [**out**] Coarse grid operator
- **op_prolong** – [**out**] Coarse to fine operator, or NULL
- **op_restrict** – [**out**] Fine to coarse operator, or NULL

Returns

An error code: 0 - success, otherwise - failure

static int **CeedBuildMassLaplace**(const *CeedScalar* *interp_1d, const *CeedScalar* *grad_1d, const *CeedScalar* *q_weight_1d, *CeedInt* P_1d, *CeedInt* Q_1d, *CeedInt* dim, *CeedScalar* *mass, *CeedScalar* *laplace)

Build 1D mass matrix and Laplacian with perturbation.

Library Developer Functions

Parameters

- **interp_1d** – [**in**] Interpolation matrix in one dimension
- **grad_1d** – [**in**] Gradient matrix in one dimension
- **q_weight_1d** – [**in**] Quadrature weights in one dimension
- **P_1d** – [**in**] Number of basis nodes in one dimension

- **`Q_1d`** – [**in**] Number of quadrature points in one dimension
- **`dim`** – [**in**] Dimension of basis
- **`mass`** – [**out**] Assembled mass matrix in one dimension
- **`laplace`** – [**out**] Assembled perturbed Laplacian in one dimension

Returns
An error code: 0 - success, otherwise - failure

7 Floating Point Precision

Currently, libCEED supports two options for `CeedScalar` : double and single. The default is to use double precision. Users wishing to set `CeedScalar` to single precision should edit `include/ceed/types.h` and change

```
#include "ceed-f64.h"  // IWYU pragma: export
```

to include `ceed-f32.h` instead, then recompile the library. Tests can be run using `make test FC=` because the Fortran tests do not support single precision at this time.

7.1 Language-specific notes

- **C**: `CEED_SCALAR_TYPE` will be defined to match one of the values of the `CeedScalarType` `enum`, and can be used for compile-time checking of `CeedScalar`'s type; see, e.g., `tests/t314-basis.c`.
- **Fortran**: There is no definition of `CeedScalar` available in the Fortran header. The user is responsible for ensuring that data used in Fortran code is of the correct type (`real*8` or `real*4`) for libCEED's current configuration.
- **Julia**: After compiling the single precision version of libCEED, instruct LibCEED.jl to use this library with the `set_libceed_path!` function and restart the Julia session. LibCEED.jl will configure itself to use the appropriate type for `CeedScalar`.
- **Python**: Make sure to replace the `ceed-f64.h` inclusion rather than commenting it out, to guarantee that the Python bindings will pick the correct precision. The `scalar_type()` function has been added to the `Ceed` class for convenience. It returns a string corresponding to a numpy datatype matching that of `CeedScalar`.
- **Rust**: The `Scalar` type corresponds to `CeedScalar`.

This is work in progress! The ability to use single precision is an initial step in ongoing development of mixed-precision support in libCEED. A current GitHub issue contains discussions related to this development.

8 Developer Notes

8.1 Style Guide

Please check your code for style issues by running

```
make format
```

In addition to those automatically enforced style rules, libCEED tends to follow the following code style conventions:

- Variable names: `snake_case`
- Strut members: `snake_case`
- Function and method names: `PascalCase` or language specific style
- Type names: `PascalCase` or language specific style
- Constant names: `CAPS_SNAKE_CASE` or language specific style

Also, documentation files should have one sentence per line to help make git diffs clearer and less disruptive.

8.2 Clang-tidy

Please check your code for common issues by running

```
make tidy
```

which uses the `clang-tidy` utility included in recent releases of Clang. This tool is much slower than actual compilation (`make -j8` parallelism helps). To run on a single file, use

```
make interface/ceed.c.tidy
```

for example. All issues reported by `make tidy` should be fixed.

8.3 Include-What-You-Use

Header inclusion for source files should follow the principal of 'include what you use' rather than relying upon transitive `#include` to define all symbols.

Every symbol that is used in the source file `foo.c` should be defined in `foo.c`, `foo.h`, or in a header file `#include`d in one of these two locations. Please check your code by running the tool `include-what-you-use` to see recommendations for changes to your source. Most issues reported by `include-what-you-use` should be fixed; however this rule is flexible to account for differences in header file organization in external libraries. If you have `include-what-you-use` installed in a sibling directory to libCEED or set the environment variable `IWYU_CC`, then you can use the makefile target `make iwyu`.

Header files should be listed in alphabetical order, with installed headers preceding local headers and `ceed` headers being listed first. The `ceed-f64.h` and `ceed-f32.h` headers should only be included in `ceed.h`.

```
#include <ceed.h>
#include <ceed/backend.h>
#include <stdbool.h>
#include <string.h>
#include "ceed-avx.h"
```

8.4 Shape

Backends often manipulate tensors of dimension greater than 2. It is awkward to pass fully-specified multi-dimensional arrays using C99 and certain operations will flatten/reshape the tensors for computational convenience. We frequently use comments to document shapes using a lexicographic ordering. For example, the comment

```
// u has shape [dim, num_comp, Q, num_elem]
```

means that it can be traversed as

```
for (d=0; d<dim; d++)
  for (c=0; c<num_comp; c++)
    for (q=0; q<Q; q++)
      for (e=0; e<num_elem; e++)
        u[((d*num_comp + c)*Q + q)*num_elem + e] = ...
```

This ordering is sometimes referred to as row-major or C-style. Note that flattening such as

```
// u has shape [dim, num_comp, Q*num_elem]
```

and

```
// u has shape [dim*num_comp, Q, num_elem]
```

are purely implicit – one just indexes the same array using the appropriate convention.

8.5 `restrict` Semantics

QFunction arguments can be assumed to have `restrict` semantics. That is, each input and output array must reside in distinct memory without overlap.

8.6 CeedVector Array Access Semantics

Backend implementations are expected to separately track 'owned' and 'borrowed' memory locations. Backends are responsible for freeing 'owned' memory; 'borrowed' memory is set by the user and backends only have read/write access to 'borrowed' memory. For any given precision and memory type, a backend should only have 'owned' or 'borrowed' memory, not both.

Backends are responsible for tracking which memory locations contain valid data. If the user calls `CeedVectorTakeArray()` on the only memory location that contains valid data, then the *CeedVector* is left in an *invalid state*. To repair an *invalid state*, the user must set valid data by calling `CeedVectorSetValue()`, `CeedVectorSetArray()`, or `CeedVectorGetArrayWrite()`.

Some checks for consistency and data validity with *CeedVector* array access are performed at the interface level. All backends may assume that array access will conform to these guidelines:

- Borrowed memory
  - *CeedVector* access to borrowed memory is set with `CeedVectorSetArray()` with `copy_mode = CEED_USE_POINTER` and revoked with `CeedVectorTakeArray()`. The user must first call `CeedVectorSetArray()` with `copy_mode = CEED_USE_POINTER` for the appropriate precision and memory type before calling `CeedVectorTakeArray()`.
  - `CeedVectorTakeArray()` cannot be called on a vector in a *invalid state*.

- Owned memory
  - Owned memory can be allocated by calling *CeedVectorSetValue()* or by calling *CeedVectorSetArray()* with copy_mode = CEED_COPY_VALUES.
  - Owned memory can be set by calling *CeedVectorSetArray()* with copy_mode = CEED_OWN_POINTER.
  - Owned memory can also be allocated by calling *CeedVectorGetArrayWrite()*. The user is responsible for manually setting the contents of the array in this case.
- Data validity
  - Internal synchronization and user calls to CeedVectorSync() cannot be made on a vector in an *invalid state*.
  - Calls to *CeedVectorGetArray()* and *CeedVectorGetArrayRead()* cannot be made on a vector in an *invalid state*.
  - Calls to *CeedVectorSetArray()* and *CeedVectorSetValue()* can be made on a vector in an *invalid state*.
  - Calls to *CeedVectorGetArrayWrite()* can be made on a vector in an *invalid* state. Data synchronization is not required for the memory location returned by *CeedVectorGetArrayWrite()*. The caller should assume that all data at the memory location returned by *CeedVectorGetArrayWrite()* is *invalid*.

8.7 Internal Layouts

Ceed backends are free to use any **E-vector** and **Q-vector** data layout, to include never fully forming these vectors, so long as the backend passes the t5** series tests and all examples. There are several common layouts for **L-vectors**, **E-vectors**, and **Q-vectors**, detailed below:

- **L-vector** layouts
  - **L-vectors** described by a *CeedElemRestriction* have a layout described by the offsets array and comp_stride parameter. Data for node i, component j, element k can be found in the **L-vector** at index offsets[i + k*elem_size] + j*comp_stride.
  - **L-vectors** described by a strided *CeedElemRestriction* have a layout described by the strides array. Data for node i, component j, element k can be found in the **L-vector** at index i*strides[0] + j*strides[1] + k*strides[2].
- **E-vector** layouts
  - If possible, backends should use *CeedElemRestrictionSetELayout()* to use the t2** tests. If the backend uses a strided **E-vector** layout, then the data for node i, component j, element k in the **E-vector** is given by i*layout[0] + j*layout[1] + k*layout[2].
  - Backends may choose to use a non-strided **E-vector** layout; however, the t2** tests will not function correctly in this case and the tests will need to be whitelisted for the backend to pass the test suite.
- **Q-vector** layouts
  - When the size of a *CeedQFunction* field is greater than 1, data for quadrature point i component j can be found in the **Q-vector** at index i + Q*j. Backends are free to provide the quadrature points in any order.
  - When the *CeedQFunction* field has emode CEED_EVAL_GRAD, data for quadrature point i, component j, derivative k can be found in the **Q-vector** at index i + Q*j + Q*size*k.

- Note that backend developers must take special care to ensure that the data in the **Q-vectors** for a field with `emode` `CEED_EVAL_NONE` is properly ordered when the backend uses different layouts for **E-vectors** and **Q-vectors**.

8.8 Backend Inheritance

There are three mechanisms by which a Ceed backend can inherit implementation from another Ceed backend. These options are set in the backend initialization routine.

1. Delegation - Developers may use *`CeedSetDelegate()`* to set a backend that will provide the implementation of any unimplemented Ceed objects.
2. Object delegation - Developers may use *`CeedSetObjectDelegate()`* to set a backend that will provide the implementation of a specific unimplemented Ceed object. Object delegation has higher precedence than delegation.
3. Operator fallback - Developers may use *`CeedSetOperatorFallbackResource()`* to set a *Ceed* resource that will provide the implementation of unimplemented *CeedOperator* methods. A fallback *Ceed* with this resource will only be instantiated if a method is called that is not implemented by the parent *Ceed*. In order to use the fallback mechanism, the parent *Ceed* and fallback resource must use compatible **E-vector** and **Q-vector** layouts.

For example, the `/cpu/self/xsmm/serial/` backend implements the `CeedTensorContract` object but delegates all other functionality to the `/cpu/self/opt/serial` backend. The `/cpu/self/opt/serial` backend implements the `CeedTensorContract` and `CeedOperator` objects but delegates all other functionality to the `/cpu/self/ref/serial` backend.

If the `/cpu/self/opt/serial` backend had missing *CeedOperator* functionality, then it could fallback to `/cpu/self/ref/serial` for missing methods. In this case, the fallback *Ceed* would clone the `/cpu/self/opt/serial` *CeedOperator* and use this clone to execute the missing functionality.

9 How to Contribute

Contributions to libCEED are encouraged.

Please make your commits well-organized and atomic, using `git rebase --interactive` as needed. Check that tests (including “examples”) pass using `make prove-all`. If adding a new feature, please add or extend a test so that your new feature is tested.

In typical development, every commit should compile, be covered by the test suite, and pass all tests. This improves the efficiency of reviewing and facilitates use of `git bisect`.

Open an issue or RFC (request for comments) pull request to discuss any significant changes before investing time. It is useful to create a WIP (work in progress) pull request for any long-running development so that others can be aware of your work and help to avoid creating merge conflicts.

Write commit messages for a reviewer of your pull request and for a future developer (maybe you) that bisects and finds that a bug was introduced in your commit. The assumptions that are clear in your mind while committing are likely not in the mind of whomever (possibly you) needs to understand it in the future.

Give credit where credit is due using tags such as `Reported-by: Helpful User <helpful@example.com>` or `Co-authored-by: Snippet Mentor <code.by@comment.com>`. Please use a real name and email for your author information (`git config user.name` and `user.email`). If your author information or email becomes inconsistent (look at `git shortlog -se`), please edit `.mailmap` to obtain your preferred name and email address.

When contributors make a major contribution and support it, their names are included in the automatically generated user-manual documentation.

Please avoid "merging from upstream" (like merging 'main' into your feature branch) unless there is a specific reason to do so, in which case you should explain why in the merge commit. Rationale from Junio and Linus.

You can use `make format` to help conform to coding conventions of the project, but try to avoid mixing whitespace or formatting changes with content changes (see atomicity above).

By submitting a pull request, you are affirming the following.

9.1 Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

9.2 Authorship

libCEED contains components authored by many individuals. It is important that contributors receive appropriate recognition through informal and academically-recognized credit systems such as publications. Status as a named author on the users manual and libCEED software publications will be granted for those who

1. make significant contributions to libCEED (in implementation, documentation, conceptualization, review, etc.) and
2. maintain and support those contributions.

Maintainers will do their best to notice when contributions reach this level and add your name to `AUTHORS`, but please email or create an issue if you believe your contributions have met these criteria and haven't yet been acknowledged.

Authors of publications about libCEED as a whole, including DOI-bearing archives, shall offer co-authorship to all individuals listed in the `AUTHORS` file. Authors of publications claiming specific libCEED contributions shall evaluate those listed in `AUTHORS` and offer co-authorship to those who made significant intellectual contributions to the work.

Note that there is no co-authorship expectation for those publishing about use of libCEED (versus creation of new features in libCEED), but see the citing section and use your judgment regarding significance of support/advice you may have received in developing your use case and interpreting results.

10 Code of Conduct

10.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

10.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

10.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

10.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

10.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at jed@jedbrown.org, valeria@caltech.edu, or tzanio@llnl.gov. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

10.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

10.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

10.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

10.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

10.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

10.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

11 Changes/Release Notes

On this page we provide a summary of the main API changes, new features and examples for each release of libCEED.

11.1 Current `main` branch

11.1.1 Interface changes

11.1.2 New features

11.1.3 Examples

11.2 v0.12 (Oct 31, 2023)

11.2.1 Interface changes

- Update `CeedOperatorContext*` functions to `CeedOperator*Context*` functions for consistency. For example, `CeedOperatorContextGetFieldLabel` was renamed to `CeedOperatorGetContextFieldLabel`.
- Removed `CeedBasisSetNumQuadraturePoints` as redundant and bug-prone interface.

11.2.2 New features

- Added *CeedOperatorGetFieldByName()* to access a specific CeedOperatorField by its name.
- Update /cpu/self/memcheck/* backends to help verify CeedVector array access assumptions and CeedQFunction user output assumptions.
- Update *CeedOperatorLinearAssembleDiagonal()* to provide default implementation that supports CeedOperator with multiple active bases.
- Added Sycl backends /gpu/sycl/ref, /gpu/sycl/shared, and /gpu/sycl/gen.
- Added *CeedBasisApplyAtPoints()* for evaluation of values and derivatives at arbitrary points inside elements.
- Added support for non-tensor $H(\text{curl})$ finite element spaces with *CeedBasisCreateHcurl()*.
- Added *CeedElemRestrictionCreateCurlOriented()*, similar to *CeedElemRestrictionCreateOriented()*, for element restrictions requiring more general element transformations such as those for high-order $H(\text{curl})$ spaces on tetrahedra (see https://dl.acm.org/doi/pdf/10.1145/3524456).
- Added *CeedOperatorLinearAssemblePointBlockDiagonalSymbolic()* to create COO mapping for mapping out of *CeedOperatorLinearAssemblePointBlockDiagonal()*.
- Added support for application codes which manage multiple *Ceed* objects, parallelized across OpenMP threads.

11.2.3 Examples

- Add DMSwarm example demonstrating interpolation from background mesh to swarm points and projection from swarm points to background mesh.

11.2.3.1 Bakeoff problems and generalizations

- Requires PETSc version 3.19 or later.

11.2.3.2 Compressible Navier-Stokes mini-app

- Updated restart and checkpointing interface.
- Add data-driven subgrid-stress model.
- Add differential filtering of solution.
- Add turbulence statistics collection over spanwise-symmetric geometries.
- Add Taylor-Green vortex initial condition.
- Add Riemann-based outflow boundary conditions.
- Added vortex shedding and flow past cylinder example, including calculations for lift, drag, and heat transfer.
- Add Internal Damping Layer (IDL) for helping turbulent simulation stability.
- Derive CeedBasis from PetscFE, and various other internal maintainability updates.

11.3 v0.11 (Dec 24, 2022)

11.3.1 Interface changes

- Added *CeedOperatorSetName()* for more readable *CeedOperatorView()* output.
- Added *CeedBasisCreateProjection()* to facilitate interpolation between nodes for separate CeedBases.
- Rename and move *CeedCompositeOperatorGetNumSub()* and *CeedCompositeOperatorGetSubList()* to public interface.
- Renamed CEED_BASIS_COLLOCATED to CEED_BASIS_NONE for clarity. Some users previously misinterpreted a CeedOperator field using CEED_BASIS_COLLOCATED as meaning that the entire CeedOperator used a quadrature space that is collocated with the nodal space of the active bases.

11.3.2 New features

- Update /cpu/self/memcheck/* backends to help verify CeedQFunctionContext data sizes provided by user.
- Improved support for $H(\mathrm{div})$ bases.
- Added CeedInt_FMT to support potential future use of larger integer sizes.
- Added CEED_QFUNCTION_ATTR for setting compiler attributes/pragmas to CEED_QFUNCTION_HELPER and CEED_QFUNCTION.
- OCCA backend updated to latest OCCA release; DPC++ and OMP OCCA modes enabled. Due to a limitation of the OCCA parser, typedefs are required to use pointers to arrays in QFunctions with the OCCA backend. This issue will be fixed in a future OCCA release.

11.3.3 Bugfix

- Fix bug in setting device id for GPU backends.
- Fix storing of indices for CeedElemRestriction on the host with GPU backends.
- Fix CeedElemRestriction sizing for CeedOperatorAssemblePointBlockDiagonal().
- Fix bugs in CPU implementation of *CeedOperatorLinearAssemble()* when there are different number of active input modes and active output modes.

11.3.4 Examples

11.3.4.1 Compressible Navier-Stokes mini-app

- Various performance enhancements, analytic matrix-free and assembled Jacobian, and PETSc solver configurations for GPUs.
- Refactored to improve code reuse and modularity.
- Support for primitive variables for more accurate boundary layers and all-speed flow.
- Added $YZ\beta$ shock capturing scheme and Shock Tube example.
- Added Channel example, with comparison to analytic solutions.

- Added Flat Plate with boundary layer mesh and compressible Blasius inflow condition based on Chebyshev collocation solution of the Blasius equations.
- Added strong and weak synthetic turbulence generation (STG) inflow boundary conditions.
- Added “freestream” boundary conditions based on HLLC Riemann solver.
- Automated stabilization coefficients for different basis degree.

11.3.4.2 Bakeoff problems and generalizations

- Support for convergence studies.

11.3.5 Maintainability

- Refactored `/gpu/cuda/shared` and `/gpu/cuda/gen` as well as `/gpu/hip/shared` and `/gpu/hip/gen` backend to improve maintainablity and reduce duplicated code.
- Enabled support for `p > 8` for `/gpu/*/shared` backends.
- Switch to `clang-format` over `astyle` for automatic formatting; Makefile command changed to `make format` from `make style`.
- Improved test harness.

11.4 v0.10.1 (Apr 11, 2022)

11.4.1 Interface changes

- Added *`CeedQFunctionSetUserFlopsEstimate()`* and *`CeedOperatorGetFlopsEstimate()`* to facilitate estimating FLOPs in operator application.

11.4.2 New features

- Switched MAGMA backends to use runtime compilation for tensor basis kernels (and element restriction kernels, in non-deterministic `/gpu/*/magma` backends). This reduces time to compile the library and increases the range of parameters for which the MAGMA tensor basis kernels will work.

11.4.3 Bugfix

- Install JiT source files in install directory to fix GPU functionality for installed libCEED.

11.5 v0.10 (Mar 21, 2022)

11.5.1 Interface changes

- Update *CeedQFunctionGetFields()* and *CeedOperatorGetFields()* to include number of fields.
- Promote to the public API: QFunction and Operator field objects, CeedQFunctionField and CeedOperatorField, and associated getters, *CeedQFunctionGetFields()*; *CeedQFunctionFieldGetName()*; *CeedQFunctionFieldGetSize()*; *CeedQFunctionFieldGetEvalMode()*; *CeedOperatorGetFields()*; *CeedOperatorFieldGetElemRestriction()*; *CeedOperatorFieldGetBasis()*; and *CeedOperatorFieldGetVector()*.
- Clarify and document conditions where CeedQFunction and CeedOperator become immutable and no further fields or suboperators can be added.
- Add *CeedOperatorLinearAssembleQFunctionBuildOrUpdate()* to reduce object creation overhead in assembly of CeedOperator preconditioning ingredients.
- Promote *CeedOperatorCheckReady()* to the public API to facilitate interactive interfaces.
- Warning added when compiling OCCA backend to alert users that this backend is experimental.
- ceed-backend.h, ceed-hash.h, and ceed-khash.h removed. Users should use ceed/backend.h, ceed/hash.h, and ceed/khash.h.
- Added *CeedQFunctionGetKernelName()*; refactored *CeedQFunctionGetSourcePath()* to exclude function kernel name.
- Clarify documentation for *CeedVectorTakeArray()*; this function will error if *CeedVectorSetArray()* with copy_mode == CEED_USE_POINTER was not previously called for the corresponding CeedMemType.
- Added *CeedVectorGetArrayWrite()* that allows access to uninitialized arrays; require initialized data for *CeedVectorGetArray()*.
- Added *CeedQFunctionContextRegisterDouble()* and *CeedQFunctionContextRegisterInt32()* with *CeedQFunctionContextSetDouble()* and *CeedQFunctionContextSetInt32()* to facilitate easy updating of *CeedQFunctionContext* data by user defined field names.
- Added CeedQFunctionContextGetFieldDescriptions() to retrieve user defined descriptions of fields that are registered with CeedQFunctionContextRegister*.
- Renamed CeedElemTopology entries for clearer namespacing between libCEED enums.
- Added type CeedSize equivalent to ptrdiff_t for array sizes in *CeedVectorCreate()*, *CeedVectorGetLength()*, CeedElemRestrictionCreate*, *CeedElemRestrictionGetLVectorSize()*, and *CeedOperatorLinearAssembleSymbolic()*. This is a breaking change.
- Added CeedOperatorSetQFunctionUpdated() to facilitate QFunction data re-use between operators sharing the same quadrature space, such as in a multigrid hierarchy.
- Added *CeedOperatorGetActiveVectorLengths()* to get shape of CeedOperator.

11.5.2 New features

- `CeedScalar` can now be set as `float` or `double` at compile time.
- Added JiT utilities in `ceed/jit-tools.h` to reduce duplicated code in GPU backends.
- Added support for JiT of QFunctions with `#include "relative/path/local-file.h"` statements for additional local files. Note that files included with `""` are searched relative to the current file first, then by compiler paths (as with `<>` includes). To use this feature, one should adhere to relative paths only, not compiler flags like `-I`, which the JiT will not be aware of.
- Remove need to guard library headers in QFunction source for code generation backends.
- `CeedDebugEnv()` macro created to provide debugging outputs when Ceed context is not present.
- Added *`CeedStringAllocCopy()`* to reduce repeated code for copying strings internally.
- Added `CeedPathConcatenate()` to facilitate loading kernel source files with a path relative to the current file.
- Added support for non-tensor $H(\mathrm{div})$ elements, to include CPU backend implementations and *`CeedBasisCreateHdiv()`* convenience constructor.
- Added *`CeedQFunctionSetContextWritable()`* and read-only access to `CeedQFunctionContext` data as an optional feature to improve GPU performance. By default, calling the `CeedQFunctionUser` during *`CeedQFunctionApply()`* is assumed to write into the `CeedQFunctionContext` data, consistent with the previous behavior. Note that if a user asserts that their `CeedQFunctionUser` does not write into the `CeedQFunctionContext` data, they are responsible for the validity of this assertion.
- Added support for element matrix assembly in GPU backends.

11.5.3 Maintainability

- Refactored preconditioner support internally to facilitate future development and improve GPU completeness/test coverage.
- `Include-what-you-use` makefile target added as `make iwyu`.
- Create backend constant `CEED_FIELD_MAX` to reduce magic numbers in codebase.
- Put GPU JiTed kernel source code into separate files.
- Dropped legacy version support in PETSc based examples to better utilize PETSc DMPlex and Mat updates to support libCEED; current minimum PETSc version for the examples is v3.17.

11.6 v0.9 (Jul 6, 2021)

11.6.1 Interface changes

- Minor modification in error handling macro to silence pedantic warnings when compiling with Clang, but no functional impact.

11.6.2 New features

- Add *CeedVectorAXPY()* and *CeedVectorPointwiseMult()* as a convenience for stand-alone testing and internal use.
- Add `CEED_QFUNCTION_HELPER` macro to properly annotate QFunction helper functions for code generation backends.
- Add `CeedPragmaOptimizeOff` macro for code that is sensitive to floating point errors from fast math optimizations.
- Rust support: split `libceed-sys` crate out of `libceed` and publish both on crates.io.

11.6.3 Performance improvements

11.6.4 Examples

- Solid mechanics mini-app updated to explore the performance impacts of various formulations in the initial and current configurations.
- Fluid mechanics example adds GPU support and improves modularity.

11.6.5 Deprecated backends

- The `/cpu/self/tmpl` and `/cpu/self/tmpl/sub` backends have been removed. These backends were intially added to test the backend inheritance mechanism, but this mechanism is now widely used and tested in multiple backends.

11.7 v0.8 (Mar 31, 2021)

11.7.1 Interface changes

- Error handling improved to include enumerated error codes for C interface return values.
- Installed headers that will follow semantic versioning were moved to `include/ceed` directory. These headers have been renamed from `ceed-*.h` to `ceed/*.h`. Placeholder headers with the old naming schema are currently provided, but these headers will be removed in the libCEED v0.9 release.

11.7.2 New features

- Julia and Rust interfaces added, providing a nearly 1-1 correspondence with the C interface, plus some convenience features.
- Static libraries can be built with `make STATIC=1` and the pkg-config file is installed accordingly.
- Add *CeedOperatorLinearAssembleSymbolic()* and *CeedOperatorLinearAssemble()* to support full assembly of libCEED operators.

11.7.3 Performance improvements

- New HIP MAGMA backends for hipMAGMA library users: `/gpu/hip/magma` and `/gpu/hip/magma/det`.
- New HIP backends for improved tensor basis performance: `/gpu/hip/shared` and `/gpu/hip/gen`.

11.7.4 Examples

- *Solid mechanics mini-app* example updated with traction boundary conditions and improved Dirichlet boundary conditions.
- *Solid mechanics mini-app* example updated with Neo-Hookean hyperelasticity in current configuration as well as improved Neo-Hookean hyperelasticity exploring storage vs computation tradeoffs.
- *Compressible Navier-Stokes mini-app* example updated with isentropic traveling vortex test case, an analytical solution to the Euler equations that is useful for testing boundary conditions, discretization stability, and order of accuracy.
- *Compressible Navier-Stokes mini-app* example updated with support for performing convergence study and plotting order of convergence by polynomial degree.

11.8 v0.7 (Sep 29, 2020)

11.8.1 Interface changes

- Replace limited `CeedInterlaceMode` with more flexible component stride `compstride` in `CeedElemRestriction` constructors. As a result, the `indices` parameter has been replaced with `offsets` and the `nnodes` parameter has been replaced with `lsize`. These changes improve support for mixed finite element methods.
- Replace various uses of `Ceed*Get*Status` with `Ceed*Is*` in the backend API to match common nomenclature.
- Replace `CeedOperatorAssembleLinearDiagonal` with *`CeedOperatorLinearAssembleDiagonal()`* for clarity.
- Linear Operators can be assembled as point-block diagonal matrices with *`CeedOperatorLinearAssemblePointBlockDiagonal()`*, provided in row-major form in a `ncomp` by `ncomp` block per node.
- Diagonal assemble interface changed to accept a *CeedVector* instead of a pointer to a *CeedVector* to reduce memory movement when interfacing with calling code.
- Added *`CeedOperatorLinearAssembleAddDiagonal()`* and *`CeedOperatorLinearAssembleAddPointBlockDiagonal()`* for improved future integration with codes such as MFEM that compose the action of *CeedOperator*s external to libCEED.
- Added `CeedVectorTakeAray()` to sync and remove libCEED read/write access to an allocated array and pass ownership of the array to the caller. This function is recommended over *`CeedVectorSyncArray()`* when the `CeedVector` has an array owned by the caller that was set by *`CeedVectorSetArray()`*.
- Added `CeedQFunctionContext` object to manage user QFunction context data and reduce copies between device and host memory.

- Added *CeedOperatorMultigridLevelCreate()*, *CeedOperatorMultigridLevelCreateTensorH1()*, and *CeedOperatorMultigridLevelCreateH1()* to facilitate creation of multigrid prolongation, restriction, and coarse grid operators using a common quadrature space.

11.8.2 New features

- New HIP backend: `/gpu/hip/ref`.
- CeedQFunction support for user `CUfunction`s in some backends

11.8.3 Performance improvements

- OCCA backend rebuilt to facilitate future performance enhancements.
- Petsc BPs suite improved to reduce noise due to multiple calls to `mpiexec`.

11.8.4 Examples

- *Solid mechanics mini-app* example updated with strain energy computation and more flexible boundary conditions.

11.8.5 Deprecated backends

- The `/gpu/cuda/reg` backend has been removed, with its core features moved into `/gpu/cuda/ref` and `/gpu/cuda/shared`.

11.9 v0.6 (Mar 29, 2020)

libCEED v0.6 contains numerous new features and examples, as well as expanded documentation in this new website.

11.9.1 New features

- New Python interface using CFFI provides a nearly 1-1 correspondence with the C interface, plus some convenience features. For instance, data stored in the `CeedVector` structure are available without copy as `numpy.ndarray`. Short tutorials are provided in Binder.
- Linear QFunctions can be assembled as block-diagonal matrices (per quadrature point, `CeedOperatorAssembleLinearQFunction()`) or to evaluate the diagonal (`CeedOperatorAssembleLinearDiagonal()`). These operations are useful for preconditioning ingredients and are used in the libCEED's multigrid examples.
- The inverse of separable operators can be obtained using *CeedOperatorCreateFDMElementInverse()* and applied with *CeedOperatorApply()*. This is a useful preconditioning ingredient, especially for Laplacians and related operators.
- New functions: *CeedVectorNorm()*, *CeedOperatorApplyAdd()*, *CeedQFunctionView()*, *CeedOperatorView()*.
- Make public accessors for various attributes to facilitate writing composable code.
- New backend: `/cpu/self/memcheck/serial`.

- QFunctions using variable-length array (VLA) pointer constructs can be used with CUDA backends. (Single source is coming soon for OCCA backends.)
- Fix some missing edge cases in CUDA backend.

11.9.2 Performance Improvements

- MAGMA backend performance optimization and non-tensor bases.
- No-copy optimization in *`CeedOperatorApply()`*.

11.9.3 Interface changes

- Replace `CeedElemRestrictionCreateIdentity` and `CeedElemRestrictionCreateBlocked` with more flexible *`CeedElemRestrictionCreateStrided()`* and *`CeedElemRestrictionCreateBlockedStrided()`*.
- Add arguments to *`CeedQFunctionCreateIdentity()`*.
- Replace ambiguous uses of `CeedTransposeMode` for L-vector identification with `CeedInterlaceMode`. This is now an attribute of the `CeedElemRestriction` (see *`CeedElemRestrictionCreate()`*) and no longer passed as `lmode` arguments to *`CeedOperatorSetField()`* and *`CeedElemRestrictionApply()`*.

11.9.4 Examples

libCEED-0.6 contains greatly expanded examples with *new documentation*. Notable additions include:

- Standalone *Ex2-Surface* (`examples/ceed/ex2-surface`): compute the area of a domain in 1, 2, and 3 dimensions by applying a Laplacian.
- PETSc *Area* (`examples/petsc/area.c`): computes surface area of domains (like the cube and sphere) by direct integration on a surface mesh; demonstrates geometric dimension different from topological dimension.
- PETSc *Bakeoff problems and generalizations*:
  - `examples/petsc/bpsraw.c` (formerly `bps.c`): transparent CUDA support.
  - `examples/petsc/bps.c` (formerly `bpsdmplex.c`): performance improvements and transparent CUDA support.
  - *Bakeoff problems on the cubed-sphere* (`examples/petsc/bpssphere.c`): generalizations of all CEED BPs to the surface of the sphere; demonstrates geometric dimension different from topological dimension.
- *Multigrid* (`examples/petsc/multigrid.c`): new p-multigrid solver with algebraic multigrid coarse solve.
- *Compressible Navier-Stokes mini-app* (`examples/fluids/navierstokes.c`; formerly `examples/navier-stokes`): unstructured grid support (using PETSc's `DMPlex`), implicit time integration, SU/SUPG stabilization, free-slip boundary conditions, and quasi-2D computational domain support.
- *Solid mechanics mini-app* (`examples/solids/elasticity.c`): new solver for linear elasticity, small-strain hyperelasticity, and globalized finite-strain hyperelasticity using p-multigrid with algebraic multigrid coarse solve.

11.10 v0.5 (Sep 18, 2019)

For this release, several improvements were made. Two new CUDA backends were added to the family of backends, of which, the new `cuda-gen` backend achieves state-of-the-art performance using single-source *CeedQFunction*. From this release, users can define Q-Functions in a single source code independently of the targeted backend with the aid of a new macro `CEED QFUNCTION` to support JIT (Just-In-Time) and CPU compilation of the user provided *CeedQFunction* code. To allow a unified declaration, the *CeedQFunction* API has undergone a slight change: the `QFunctionField` parameter `ncomp` has been changed to `size`. This change requires setting the previous value of `ncomp` to `ncomp*dim` when adding a `QFunctionField` with eval mode `CEED EVAL GRAD`.

Additionally, new CPU backends were included in this release, such as the `/cpu/self/opt/*` backends (which are written in pure C and use partial **E-vectors** to improve performance) and the `/cpu/self/ref/memcheck` backend (which relies upon the Valgrind Memcheck tool to help verify that user *CeedQFunction* have no undefined values). This release also included various performance improvements, bug fixes, new examples, and improved tests. Among these improvements, vectorized instructions for *CeedQFunction* code compiled for CPU were enhanced by using `CeedPragmaSIMD` instead of `CeedPragmaOMP`, implementation of a *CeedQFunction* gallery and identity Q-Functions were introduced, and the PETSc benchmark problems were expanded to include unstructured meshes handling were. For this expansion, the prior version of the PETSc BPs, which only included data associated with structured geometries, were renamed `bpsraw`, and the new version of the BPs, which can handle data associated with any unstructured geometry, were called `bps`. Additionally, other benchmark problems, namely BP2 and BP4 (the vector-valued versions of BP1 and BP3, respectively), and BP5 and BP6 (the collocated versions—for which the quadrature points are the same as the Gauss Lobatto nodes—of BP3 and BP4 respectively) were added to the PETSc examples. Furthermoew, another standalone libCEED example, called `ex2`, which computes the surface area of a given mesh was added to this release.

Backends available in this release:

CEED resource (-ceed)	Backend
`/cpu/self/ref/serial`	Serial reference implementation
`/cpu/self/ref/blocked`	Blocked reference implementation
`/cpu/self/ref/memcheck`	Memcheck backend, undefined value checks
`/cpu/self/opt/serial`	Serial optimized C implementation
`/cpu/self/opt/blocked`	Blocked optimized C implementation
`/cpu/self/avx/serial`	Serial AVX implementation
`/cpu/self/avx/blocked`	Blocked AVX implementation
`/cpu/self/xsmm/serial`	Serial LIBXSMM implementation
`/cpu/self/xsmm/blocked`	Blocked LIBXSMM implementation
`/cpu/occa`	Serial OCCA kernels
`/gpu/occa`	CUDA OCCA kernels
`/omp/occa`	OpenMP OCCA kernels
`/ocl/occa`	OpenCL OCCA kernels
`/gpu/cuda/ref`	Reference pure CUDA kernels
`/gpu/cuda/reg`	Pure CUDA kernels using one thread per element
`/gpu/cuda/shared`	Optimized pure CUDA kernels using shared memory
`/gpu/cuda/gen`	Optimized pure CUDA kernels using code generation
`/gpu/magma`	CUDA MAGMA kernels

Examples available in this release:

User code	Example
`ceed`	• ex1 (volume) • ex2 (surface)
`mfem`	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)
`petsc`	• BP1 (scalar mass operator) • BP2 (vector mass operator) • BP3 (scalar Laplace operator) • BP4 (vector Laplace operator) • BP5 (collocated scalar Laplace operator) • BP6 (collocated vector Laplace operator) • Navier-Stokes
`nek5000`	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)

11.11 v0.4 (Apr 1, 2019)

libCEED v0.4 was made again publicly available in the second full CEED software distribution, release CEED 2.0. This release contained notable features, such as four new CPU backends, two new GPU backends, CPU backend optimizations, initial support for operator composition, performance benchmarking, and a Navier-Stokes demo. The new CPU backends in this release came in two families. The `/cpu/self/*/serial` backends process one element at a time and are intended for meshes with a smaller number of high order elements. The `/cpu/self/*/blocked` backends process blocked batches of eight interlaced elements and are intended for meshes with higher numbers of elements. The `/cpu/self/avx/*` backends rely upon AVX instructions to provide vectorized CPU performance. The `/cpu/self/xsmm/*` backends rely upon the LIBXSMM package to provide vectorized CPU performance. The `/gpu/cuda/*` backends provide GPU performance strictly using CUDA. The `/gpu/cuda/ref` backend is a reference CUDA backend, providing reasonable performance for most problem configurations. The `/gpu/cuda/reg` backend uses a simple parallelization approach, where each thread treats a finite element. Using just in time compilation, provided by nvrtc (NVidia Runtime Compiler), and runtime parameters, this backend unroll loops and map memory address to registers. The `/gpu/cuda/reg` backend achieve good peak performance for 1D, 2D, and low order 3D problems, but performance deteriorates very quickly when threads run out of registers.

A new explicit time-stepping Navier-Stokes solver was added to the family of libCEED examples in the `examples/petsc` directory (see *Compressible Navier-Stokes mini-app*). This example solves the time-dependent Navier-Stokes equations of compressible gas dynamics in a static Eulerian three-dimensional frame, using structured high-order finite/spectral element spatial discretizations and explicit high-order time-stepping (available in PETSc). Moreover, the Navier-Stokes example was developed using PETSc, so that the pointwise physics (defined at quadrature points) is separated from the parallelization and meshing concerns.

Backends available in this release:

CEED resource (-ceed)	Backend
/cpu/self/ref/serial	Serial reference implementation
/cpu/self/ref/blocked	Blocked reference implementation
/cpu/self/tmpl	Backend template, defaults to /cpu/self/blocked
/cpu/self/avx/serial	Serial AVX implementation
/cpu/self/avx/blocked	Blocked AVX implementation
/cpu/self/xsmm/serial	Serial LIBXSMM implementation
/cpu/self/xsmm/blocked	Blocked LIBXSMM implementation
/cpu/occa	Serial OCCA kernels
/gpu/occa	CUDA OCCA kernels
/omp/occa	OpenMP OCCA kernels
/ocl/occa	OpenCL OCCA kernels
/gpu/cuda/ref	Reference pure CUDA kernels
/gpu/cuda/reg	Pure CUDA kernels using one thread per element
/gpu/magma	CUDA MAGMA kernels

Examples available in this release:

User code	Example
ceed	• ex1 (volume)
mfem	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)
petsc	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator) • Navier-Stokes
nek5000	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)

11.12 v0.3 (Sep 30, 2018)

Notable features in this release include active/passive field interface, support for non-tensor bases, backend optimization, and improved Fortran interface. This release also focused on providing improved continuous integration, and many new tests with code coverage reports of about 90%. This release also provided a significant change to the public interface: a *CeedQFunction* can take any number of named input and output arguments while *CeedOperator* connects them to the actual data, which may be supplied explicitly to `CeedOperatorApply()` (active) or separately via `CeedOperatorSetField()` (passive). This interface change enables reusable libraries of CeedQFunctions and composition of block solvers constructed using *CeedOperator*. A concept of blocked restriction was added to this release and used in an optimized CPU backend. Although this is typically not visible to the user, it enables effective use of arbitrary-length SIMD while maintaining cache locality. This CPU backend also implements an algebraic factorization of tensor product gradients to perform fewer operations than standard application of interpolation and differentiation from nodes to quadrature points. This algebraic formulation automatically supports non-polynomial and non-interpolatory bases, thus is more general than the more common derivation in terms of Lagrange polynomials on the quadrature points.

Backends available in this release:

CEED resource (-ceed)	Backend
/cpu/self/blocked	Blocked reference implementation
/cpu/self/ref	Serial reference implementation
/cpu/self/tmpl	Backend template, defaults to /cpu/self/blocked
/cpu/occa	Serial OCCA kernels
/gpu/occa	CUDA OCCA kernels
/omp/occa	OpenMP OCCA kernels
/ocl/occa	OpenCL OCCA kernels
/gpu/magma	CUDA MAGMA kernels

Examples available in this release:

User code	Example
ceed	• ex1 (volume)
mfem	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)
petsc	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)
nek5000	• BP1 (scalar mass operator) • BP3 (scalar Laplace operator)

11.13 v0.21 (Sep 30, 2018)

A MAGMA backend (which relies upon the MAGMA package) was integrated in libCEED for this release. This initial integration set up the framework of using MAGMA and provided the libCEED functionality through MAGMA kernels as one of libCEED's computational backends. As any other backend, the MAGMA backend provides extended basic data structures for *CeedVector*, *CeedElemRestriction*, and *CeedOperator*, and implements the fundamental CEED building blocks to work with the new data structures. In general, the MAGMA-specific data structures keep the libCEED pointers to CPU data but also add corresponding device (e.g., GPU) pointers to the data. Coherency is handled internally, and thus seamlessly to the user, through the functions/methods that are provided to support them.

Backends available in this release:

CEED resource (-ceed)	Backend
/cpu/self	Serial reference implementation
/cpu/occa	Serial OCCA kernels
/gpu/occa	CUDA OCCA kernels
/omp/occa	OpenMP OCCA kernels
/ocl/occa	OpenCL OCCA kernels
/gpu/magma	CUDA MAGMA kernels

Examples available in this release:

User code	Example
`ceed`	- ex1 (volume)
`mfem`	- BP1 (scalar mass operator) - BP3 (scalar Laplace operator)
`petsc`	- BP1 (scalar mass operator)
`nek5000`	- BP1 (scalar mass operator)

11.14 v0.2 (Mar 30, 2018)

libCEED was made publicly available the first full CEED software distribution, release CEED 1.0. The distribution was made available using the Spack package manager to provide a common, easy-to-use build environment, where the user can build the CEED distribution with all dependencies. This release included a new Fortran interface for the library. This release also contained major improvements in the OCCA backend (including a new `/ocl/occa` backend) and new examples. The standalone libCEED example was modified to compute the volume volume of a given mesh (in 1D, 2D, or 3D) and placed in an `examples/ceed` subfolder. A new `mfem` example to perform BP3 (with the application of the Laplace operator) was also added to this release.

Backends available in this release:

CEED resource (`-ceed`)	Backend
`/cpu/self`	Serial reference implementation
`/cpu/occa`	Serial OCCA kernels
`/gpu/occa`	CUDA OCCA kernels
`/omp/occa`	OpenMP OCCA kernels
`/ocl/occa`	OpenCL OCCA kernels

Examples available in this release:

User code	Example
`ceed`	- ex1 (volume)
`mfem`	- BP1 (scalar mass operator) - BP3 (scalar Laplace operator)
`petsc`	- BP1 (scalar mass operator)
`nek5000`	- BP1 (scalar mass operator)

11.15 v0.1 (Jan 3, 2018)

Initial low-level API of the CEED project. The low-level API provides a set of Finite Elements kernels and components for writing new low-level kernels. Examples include: vector and sparse linear algebra, element matrix assembly over a batch of elements, partial assembly and action for efficient high-order operators like mass, diffusion, advection, etc. The main goal of the low-level API is to establish the basis for the high-level API. Also, identifying such low-level kernels and providing a reference implementation for them serves as the basis for specialized backend implementations. This release contained several backends: `/cpu/self`, and backends which rely upon the OCCA package, such as `/cpu/occa`, `/gpu/occa`, and `/omp/occa`. It also included several examples, in the `examples` folder: A standalone code that shows the usage of libCEED (with no external dependencies) to apply the Laplace operator, `ex1`; an `mfem` example to perform BP1 (with the application of the mass operator); and a `petsc` example to perform BP1 (with the application of the mass operator).

Backends available in this release:

CEED resource (-ceed)	Backend
`/cpu/self`	Serial reference implementation
`/cpu/occa`	Serial OCCA kernels
`/gpu/occa`	CUDA OCCA kernels
`/omp/occa`	OpenMP OCCA kernels

Examples available in this release:

User code	Example
`ceed`	ex1 (scalar Laplace operator)
`mfem`	BP1 (scalar mass operator)
`petsc`	BP1 (scalar mass operator)

12 Indices and tables

- genindex
- search

References

[sod] Sod shock tube. https://en.wikipedia.org/wiki/Sod_shock_tube. Accessed: 01-30-2022.

[AB93] Ellen M Arruda and Mary C Boyce. A three-dimensional constitutive model for the large stretch behavior of rubber elastic materials. *Journal of the Mechanics and Physics of Solids*, 41(2):389–412, 1993. doi:10.1016/0022-5096(93)90013-6.

[BAB+21] Jed Brown, Ahmad Abdelfattah, Valeria Barra, Natalie Beams, Jean Sylvain Camier, Veselin Dobrev, Yohann Dudouit, Leila Ghaffari, Tzanio Kolev, David Medina, Will Pazner, Thilina Ratnayaka, Jeremy Thompson, and Stan Tomov. libCEED: fast algebra for high-order element-based discretizations. *Journal of Open Source Software*, 6(63):2945, 2021. doi:10.21105/joss.02945.

[BJ16] Jonathan R. Bull and Antony Jameson. Explicit filtering and exact reconstruction of the sub-filter stresses in large eddy simulation. *Journal of Computational Physics*, 306:117–136, 2016. doi:10.1016/j.jcp.2015.11.037.

[Col23] Tim Colonius. Chapter 8 - boundary conditions for turbulence simulation. In Robert D. Moser, editor, *Numerical Methods in Turbulence Simulation*, Numerical Methods in Turbulence, pages 319–357. Academic Press, 2023. doi:10.1016/B978-0-32-391144-3.00014-0.

[DPA+20] Denis Davydov, Jean-Paul Pelteret, Daniel Arndt, Martin Kronbichler, and Paul Steinmann. A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid. *International Journal for Numerical Methods in Engineering*, 121(13):2874–2895, 2020. doi:10.1002/nme.6336.

[Ger86] M. Germano. Differential filters for the large eddy numerical simulation of turbulent flows. *The Physics of Fluids*, 29(6):1755–1757, 1986. doi:10.1063/1.865649.

[Hol00] Gerhard Holzapfel. *Nonlinear solid mechanics: a continuum approach for engineering*. Wiley, Chichester New York, 2000. ISBN 978-0-471-82319-3.

[HST10] Thomas J R Hughes, Guglielmo Scovazzi, and Tayfun E Tezduyar. Stabilized methods for compressible flows. *Journal of Scientific Computing*, 43:343–368, 2010. doi:10.1007/s10915-008-9233-5.

[Hug12] Thomas JR Hughes. *The finite element method: linear static and dynamic finite element analysis*. Courier Corporation, 2012.

[MDGP+14] Gianmarco Mengaldo, Daniele De Grazia, Joaquim Peiro, Antony Farrington, Freddie Witherden, Peter Vincent, and Spencer Sherwin. A guide to the implementation of boundary conditions in compact high-order methods for compressible aerodynamics. In *AIAA Aviation 2014*. Atlanta, June 2014. AIAA. doi:10.2514/6.2014-2923.

[PMK92] TC Papanastasiou, N Malamataris, and Ellwood K. A new outflow boundary condition. *International Journal for Numerical Methods in Fluids*, 14:587–608, March 1992. doi:10.1002/fld.1650140506.

[Pop00] Stephen B Pope. *Turbulent Flows*. Cambridge University Press, 2000. ISBN 9780521598866.

[PJE22a] Aviral Prakash, Kenneth E. Jansen, and John A. Evans. Invariant data-driven subgrid stress modeling in the strain-rate eigenframe for large eddy simulation. *Computer Methods in Applied Mechanics and Engineering*, September 2022. doi:10.1016/j.cma.2022.115457.

[PJE22b] Aviral Prakash, Kenneth E. Jansen, and John A. Evans. Invariant data-driven subgrid stress modeling on anisotropic grids for large eddy simulation. 2022. arXiv:arXiv:2212.00332.

[SHJ91] Farzin Shakib, Thomas JR Hughes, and Zdeněk Johan. A new finite element formulation for computational fluid dynamics: X. the compressible Euler and Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 89(1-3):141–219, 1991. doi:10.1016/0045-7825(91)90041-4.

[SSST14] Michael L. Shur, Philippe R. Spalart, Michael K. Strelets, and Andrey K. Travin. Synthetic turbulence generators for RANS-LES interfaces in zonal simulations of aerodynamic and aeroacoustic problems. *Flow, Turbulence and Combustion*, 93(1):63–92, 2014. doi:10.1007/s10494-014-9534-8.

[SWW+93] Jerry M Straka, Robert B Wilhelmson, Louis J Wicker, John R Anderson, and Kelvin K Droegemeier. Numerical solutions of a non-linear density current: a benchmark solution and comparisons. *International Journal for Numerical Methods in Fluids*, 17(1):1–22, 1993. doi:10.1002/fld.1650170103.

[TS07] Tayfun E Tezduyar and Masayoshi Senga. SUPG finite element computation of inviscid supersonic flows with $yz\beta $ shock capturing. *Computers and Fluids*, 36(1):147–159, 2007. doi:10.1016/j.compfluid.2005.07.009.

[Tor09] Eleuterio F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-540-49834-6.

[VD56] E. R. Van Driest. On turbulent flow near a wall. *Journal of the Aeronautical Sciences*, 23(11):1007–1011, November 1956. doi:10/ghbxk3.

[Whi99] Christian H Whiting. *Stabilized Finite Element Methods for Fluid Dynamics Using a Hierarchical Basis*. PhD thesis, Rennselear Polytechnic Institute, Troy, NY, 1999.

[WJD03] Christian H Whiting, Kenneth E Jansen, and Saikat Dey. Hierarchical basis for stabilized finite element methods for compressible flows. *Computer Methods in Applied Mechanics and Engineering*, 192(47-48):5167–5185, 2003. doi:10.1016/j.cma.2003.07.011.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi:10.1145/1498765.1498785.

[ZZS11] Rui Zhang, Mengping Zhang, and Chi-Wang Shu. On the order of accuracy and numerical performance of two classes of finite volume weno schemes. *Communications in Computational Physics*, 9(3):807–827, 2011. doi:10.4208/cicp.291109.080410s.

[Brown10] J. Brown. Efficient Nonlinear Solvers for Nodal High-Order Finite Elements in 3D. *Journal of Scientific Computing*, October 2010. doi:10.1007/s10915-010-9396-8.

Index