CENTER FOR EFFICIENT EXASCALE DISCRETIZATIONS

libCEED User Manual

Release 0.7

Ahmad Abdelfattah Valeria Barra Natalie Beams Jed Brown Jean-Sylvain Camier Veselin Dobrev Yohann Dudouit Leila Ghaffari Tzanio Kolev David Medina Will Pazner Thilina Rathnayake Jeremy L. Thompson Stan Tomov

Dec 30, 2020

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 they 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 right 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.

Furthermore, software packages that provide high-performance implementations have often been special-purpose and intrusive. libCEED 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

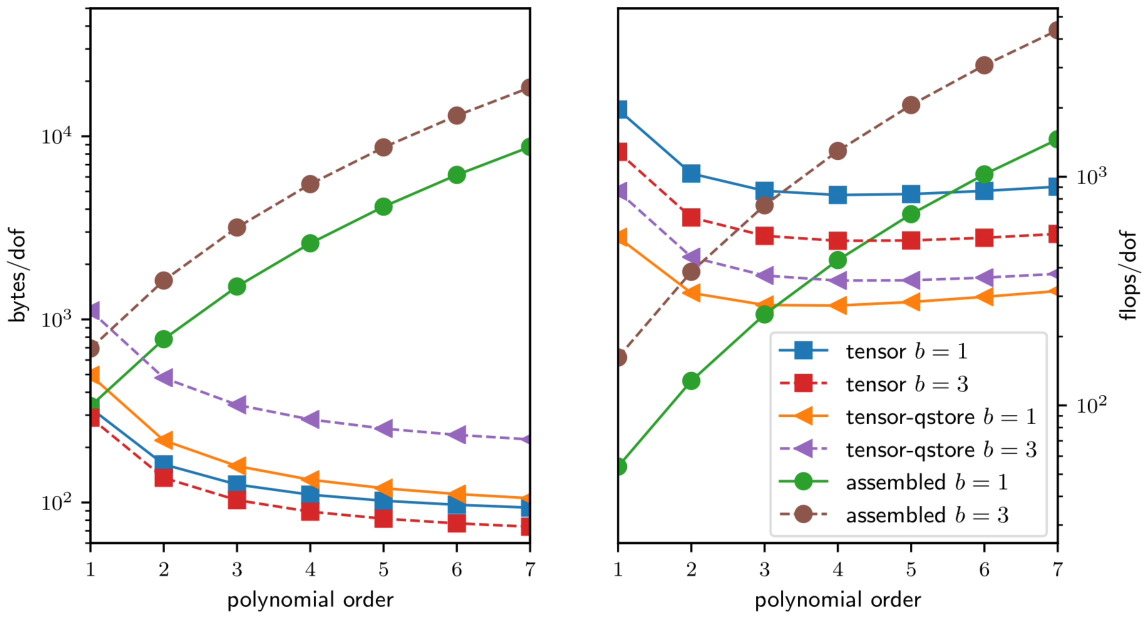


Fig. 1.1: Comparison of memory transfer and floating point operations per degree of freedom for different representations of a linear operator for a PDE in 3D with b components and variable coefficients arising due to Newton linearization of a material nonlinearity. The representation labeled as *tensor* 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.

are available via pure CUDA 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.

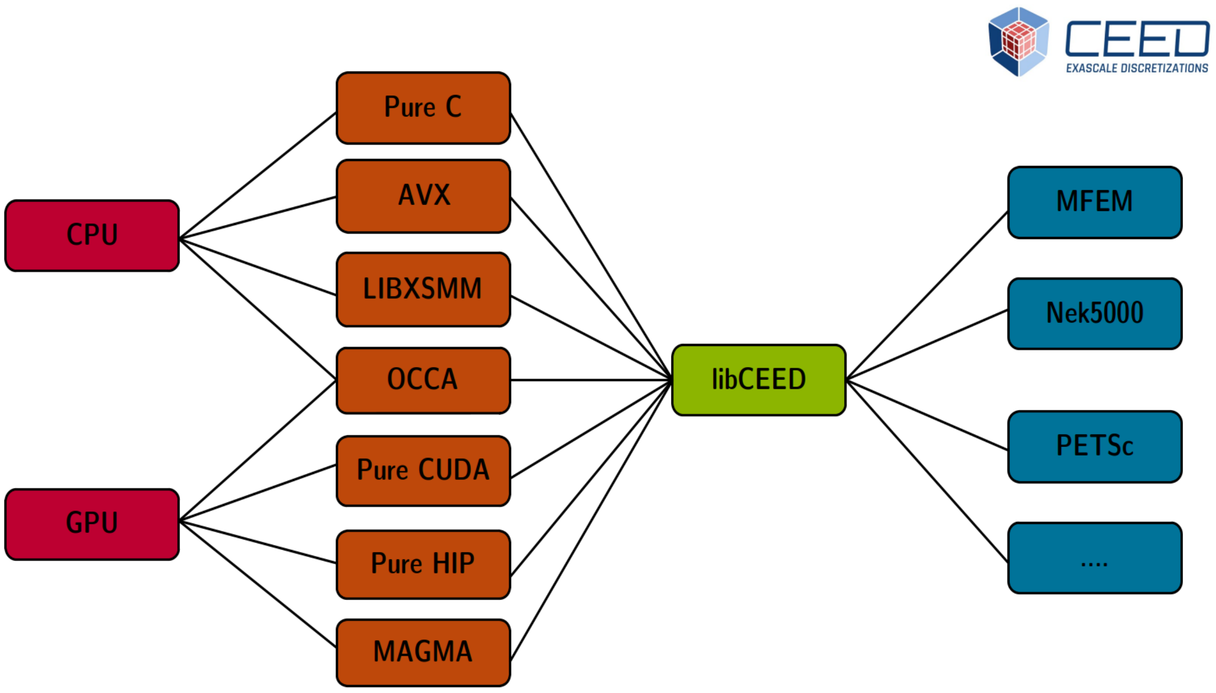


Fig. 1.2: The role of libCEED as a lightweight, portable library which provides a low-level API for efficient, specialized implementations. libCEED allows different applications to share highly optimized discretization kernels.

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.

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
```

in the Julia package manager or in a clone of the repository via:

```
JULIA_LIBCEED_LIB=/path/to/libceed.so julia
julia> # press ] to enter package manager
(env) pkg> build LibCEED
```

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

```
[dependencies]
libceed = { git = "https://github.com/CEED/libCEED", branch = "main" }
```

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 Backends		
`/cpu/self/ref/serial`	Serial reference implementation	Yes
`/cpu/self/ref/blocked`	Blocked reference implementation	Yes

continues on next page

Table 2.1 – continued from previous page

`/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 Backends		
`/cpu/self/memcheck/*`	Memcheck backends, undefined value checks	Yes
CPU LIBXSMM Backends		
`/cpu/self/xsmm/serial`	Serial LIBXSMM implementation	Yes
`/cpu/self/xsmm/blocked`	Blocked LIBXSMM implementation	Yes
CUDA Native Backends		
`/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 Backends		
`/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
MAGMA Backends		
`/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
OCCA Backends		
`/*/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
`/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 3.5 or newer is required.

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`.

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`).

Additionally, 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
```

(continues on next page)

(continued from previous page)

```
./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
./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␣
↪linElas -forcing mms
./elasticity -ceed /gpu/cuda -mesh [.exo file] -degree 2 -E 1 -nu 0.3 -problem␣
↪linElas -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 OCCA backend 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=/usr/local
```

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

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

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:

```
@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
                   Rathnayake, Thilina and
                   Thompson, Jeremy L and
                   Tomov, Stanimire},
  title        = {libCEED User Manual},
  month        = sep,
  year         = 2020,
  publisher    = {Zenodo},
  version      = {0.7},
  doi          = {10.5281/zenodo.4302737},
  url          = {https://doi.org/10.5281/zenodo.4302737}
}
```

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␣
↪and {D}avid {S}hupe},
  doi       = {10.25080/Majora-342d178e-00c},
  url       = {https://doi.org/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, Lawrence Livermore National Security, LLC. Produced at the Lawrence Livermore National Laboratory. LLNL-CODE-734707. 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 $\nabla \boldsymbol{v} : \boldsymbol{f}_1$ 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 $\boldsymbol{f}_0$ directly into v, and the term $\boldsymbol{f}_1$ directly into dv (which stands for $\nabla \boldsymbol{v}$). If equation (3.1) only presents a term of the type $\boldsymbol{f}_0$, the *CeedQFunction* will only have one output argument, namely v. If equation (3.1) also presents a term of the type $\boldsymbol{f}_1$, 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 $\boldsymbol{G}$
- 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 $\boldsymbol{P}$, $\boldsymbol{G}$ and $\boldsymbol{B}$.

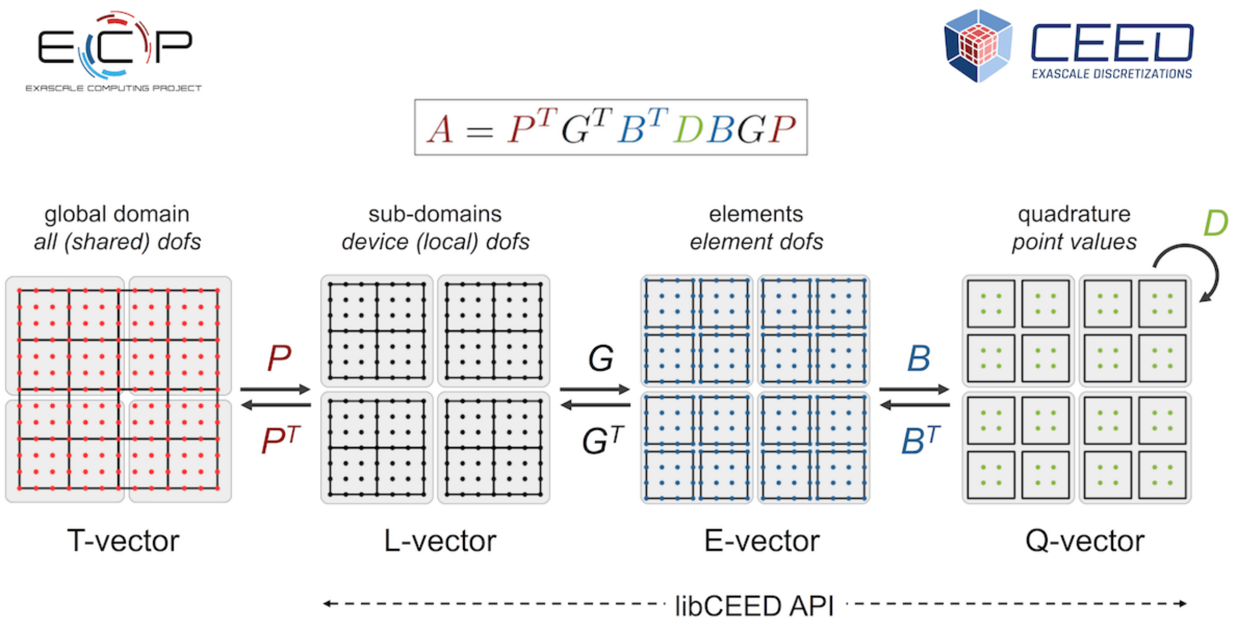


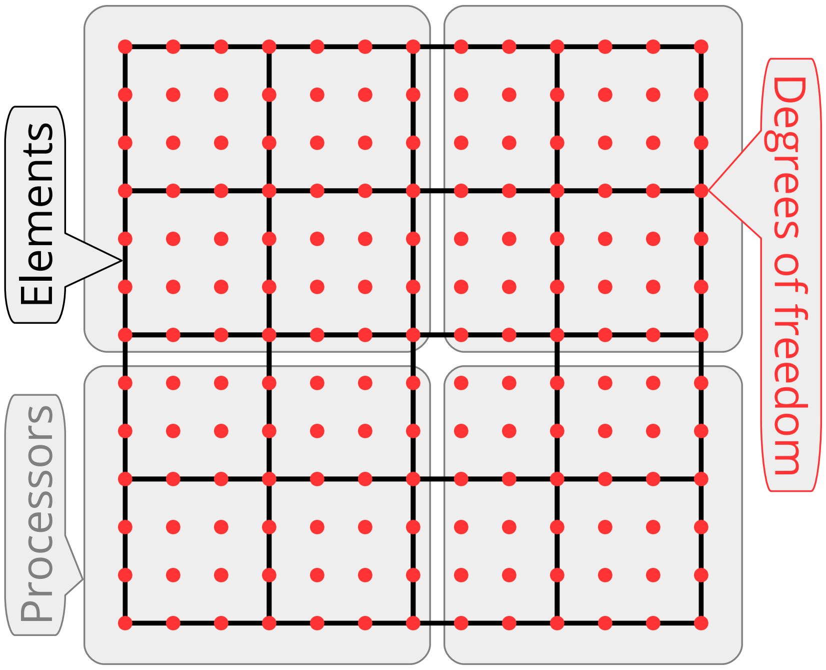
Fig. 3.1: Operator Decomposition

Note that in the case of adaptive mesh refinement (AMR), the restrictions $\boldsymbol{P}$ and $\boldsymbol{G}$ 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.).

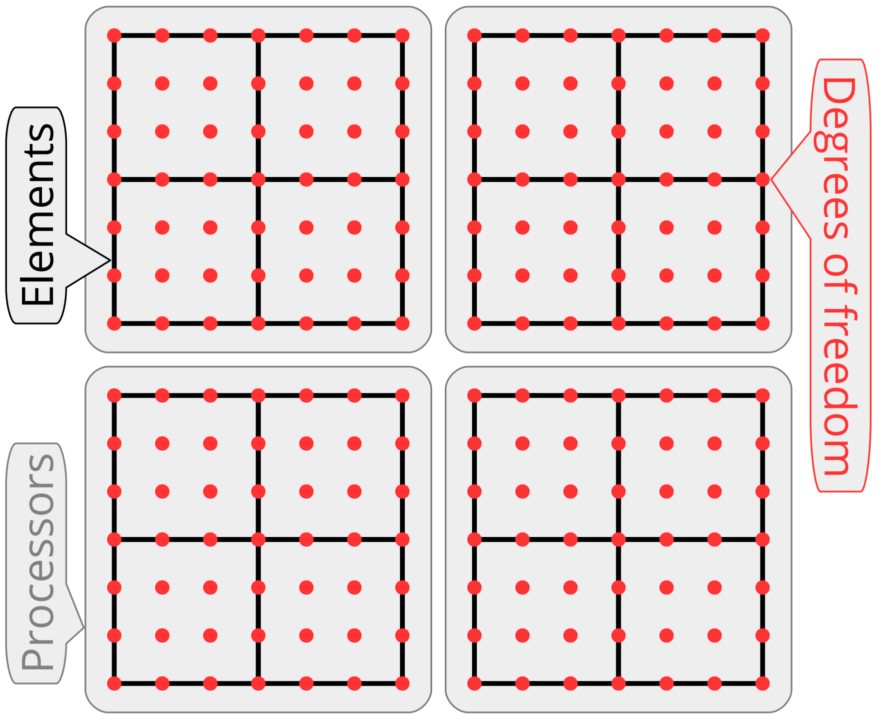
3.2.1 Terminology and Notation

Vector representation/storage categories:

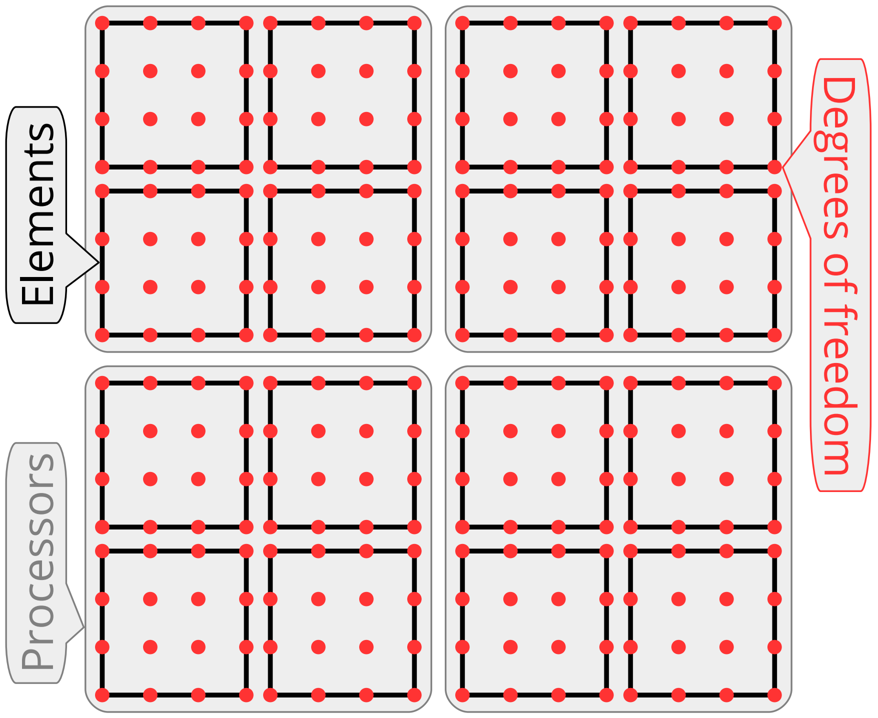
- True degrees of freedom/unknowns, **T-vector**:
  - each unknown i has exactly one copy, on exactly one processor, $rank(i)$
  - this is a non-overlapping vector decomposition
  - usually includes any essential (fixed) dofs.



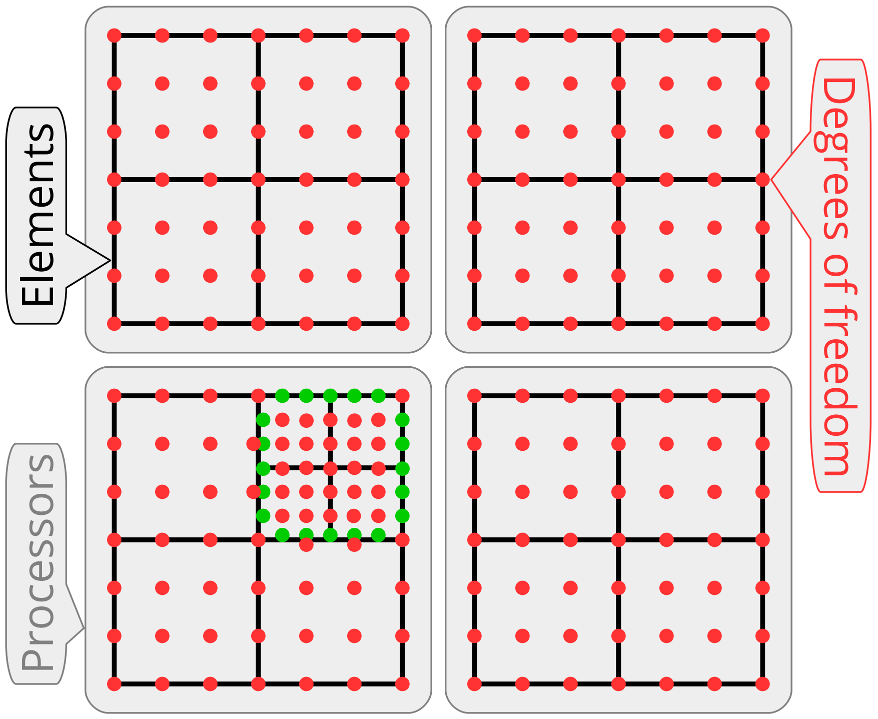
- Local (w.r.t. processors) degrees of freedom/unknowns, **L-vector**:
  - each unknown i has exactly one copy on each processor that owns an element containing i
  - this is an overlapping vector decomposition with overlaps only across different processors—there is no duplication of unknowns on a single 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 $H(\mathrm{div})$—and $H(\mathrm{curl})$-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, $\boldsymbol{P}$, (and its transpose) should use optimized matrix-free action
  - note that $\boldsymbol{P}$ 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 $\boldsymbol{B}$, $\boldsymbol{G}$ 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 $\boldsymbol{G}$. 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 $\boldsymbol{P}$, $\boldsymbol{G}$ 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, $\boldsymbol{P}$, $\boldsymbol{G}$ and $\boldsymbol{B}$, the operator evaluation is decoupled on their ranges, so $\boldsymbol{P}$, $\boldsymbol{G}$ 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 $\boldsymbol{P}$, $\boldsymbol{G}$, $\boldsymbol{B}$ and $\boldsymbol{D}$ clearly separate the MPI parallelism in the operator ($\boldsymbol{P}$) from the unstructured mesh topology ($\boldsymbol{G}$), 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 $\boldsymbol{P}$, sparse (on-device) linear algebra for $\boldsymbol{G}$, dense/structured linear algebra (tensor contractions) for $\boldsymbol{B}$ and parallel point-wise evaluations for $\boldsymbol{D}$.

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 $\boldsymbol{P}$, 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 **G**, **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).

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**.)
- $\boldsymbol{G}$ 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 $\boldsymbol{G}^T\boldsymbol{B}^T\boldsymbol{D}\boldsymbol{B}\boldsymbol{G}$ 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).

```
/// @file
/// Test creation, action, and destruction for mass matrix operator
/// \test Test creation, action, and destruction for mass matrix operator
#include <ceed.h>
#include <stdlib.h>
#include <math.h>

#include "t500-operator.h"

int main(int argc, char **argv) {
  Ceed ceed;
  CeedElemRestriction Erestrictx, Erestrictu, Erestrictui;
  CeedBasis bx, bu;
  CeedQFunction qf_setup, qf_mass;
  CeedOperator op_setup, op_mass;
  CeedVector qdata, X, U, V;
  const CeedScalar *hv;
  CeedInt nelem = 15, P = 5, Q = 8;
  CeedInt Nx = nelem+1, Nu = nelem*(P-1)+1;
  CeedInt indx[nelem*2], indu[nelem*P];
  CeedScalar x[Nx];

//! [Ceed Init]
  CeedInit(argv[1], &ceed);
//! [Ceed Init]
  for (CeedInt i=0; i<Nx; i++)
    x[i] = (CeedScalar) i / (Nx - 1);
  for (CeedInt i=0; i<nelem; i++) {
    indx[2*i+0] = i;
    indx[2*i+1] = i+1;
  }
//! [ElemRestr Create]
  CeedElemRestrictionCreate(ceed, nelem, 2, 1, 1, Nx, CEED_MEM_HOST,
                            CEED_USE_POINTER, indx, &Erestrictx);
//! [ElemRestr Create]

  for (CeedInt i=0; i<nelem; i++) {
    for (CeedInt j=0; j<P; j++) {
      indu[P*i+j] = i*(P-1) + j;
    }
  }
//! [ElemRestrU Create]
  CeedElemRestrictionCreate(ceed, nelem, P, 1, 1, Nu, CEED_MEM_HOST,
                            CEED_USE_POINTER, indu, &Erestrictu);
  CeedInt stridesu[3] = {1, Q, Q};
  CeedElemRestrictionCreateStrided(ceed, nelem, Q, 1, Q*nelem, stridesu,
                                   &Erestrictui);
//! [ElemRestrU Create]

//! [Basis Create]
```

(continues on next page)

(continued from previous page)

```
  CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, 2, Q, CEED_GAUSS, &bx);
  CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, P, Q, CEED_GAUSS, &bu);
//! [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, Nx, &X);
  CeedVectorSetArray(X, CEED_MEM_HOST, CEED_USE_POINTER, x);
  CeedVectorCreate(ceed, nelem*Q, &qdata);

//! [Setup Set]
  CeedOperatorSetField(op_setup, "_weight", CEED_ELEMRESTRICTION_NONE, bx,
                       CEED_VECTOR_NONE);
  CeedOperatorSetField(op_setup, "dx", Erestrictx, bx, CEED_VECTOR_ACTIVE);
  CeedOperatorSetField(op_setup, "rho", Erestrictui, CEED_BASIS_COLLOCATED,
                       CEED_VECTOR_ACTIVE);
//! [Setup Set]

//! [Operator Set]
  CeedOperatorSetField(op_mass, "rho", Erestrictui,CEED_BASIS_COLLOCATED,
                       qdata);
  CeedOperatorSetField(op_mass, "u", Erestrictu, bu, CEED_VECTOR_ACTIVE);
  CeedOperatorSetField(op_mass, "v", Erestrictu, bu, CEED_VECTOR_ACTIVE);
//! [Operator Set]

//! [Setup Apply]
  CeedOperatorApply(op_setup, X, qdata, CEED_REQUEST_IMMEDIATE);
//! [Setup Apply]

  CeedVectorCreate(ceed, Nu, &U);
  CeedVectorSetValue(U, 0.0);
  CeedVectorCreate(ceed, Nu, &V);
//! [Operator Apply]
  CeedOperatorApply(op_mass, U, V, CEED_REQUEST_IMMEDIATE);
//! [Operator Apply]

```

(continues on next page)

(continued from previous page)

```
  CeedVectorGetArrayRead(V, CEED_MEM_HOST, &hv);
  for (CeedInt i=0; i<Nu; i++)
    if (fabs(hv[i]) > 1e-14) printf("[%d] v %g != 0.0\n",i, hv[i]);
  CeedVectorRestoreArrayRead(V, &hv);

  CeedQFunctionDestroy(&qf_setup);
  CeedQFunctionDestroy(&qf_mass);
  CeedOperatorDestroy(&op_setup);
  CeedOperatorDestroy(&op_mass);
  CeedElemRestrictionDestroy(&Erestrictu);
  CeedElemRestrictionDestroy(&Erestrictx);
  CeedElemRestrictionDestroy(&Erestrictui);
  CeedBasisDestroy(&bu);
  CeedBasisDestroy(&bx);
  CeedVectorDestroy(&X);
  CeedVectorDestroy(&U);
  CeedVectorDestroy(&V);
  CeedVectorDestroy(&qdata);
  CeedDestroy(&ceed);
  return 0;
}
```

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.

The `setup` routine above computes and stores $\boldsymbol{D}$, in this case a scalar value in each quadrature point, while `mass` uses these saved values to perform the action of $\boldsymbol{D}$. 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 $\boldsymbol{B}$ operators for the mesh nodes, `bx`, and the unknown field, `bu`, 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, &bx);
CeedBasisCreateTensorH1Lagrange(ceed, 1, 1, P, Q, CEED_GAUSS, &bu);
```

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 $\boldsymbol{G}$ operators for the mesh nodes, `Erestrictx`, and the unknown field, `Erestrictu`, are specified in the *`CeedElemRestrictionCreate()`*. Both of these specify directly the dof indices for each element in the `indx` and `indu` arrays:

```
CeedElemRestrictionCreate(ceed, nelem, 2, 1, 1, Nx, CEED_MEM_HOST,
                          CEED_USE_POINTER, indx, &Erestrictx);
```

```
CeedElemRestrictionCreate(ceed, nelem, P, 1, 1, Nu, CEED_MEM_HOST,
                          CEED_USE_POINTER, indu, &Erestrictu);
CeedInt stridesu[3] = {1, Q, Q};
CeedElemRestrictionCreateStrided(ceed, nelem, Q, 1, Q*nelem, stridesu,
                                 &Erestrictui);
```

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 $\boldsymbol{G}$ is the identity and *`CeedElemRestrictionCreateStrided()`* is used instead. We plan to support other structured representations of $\boldsymbol{G}$ 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 $\boldsymbol{G}$ 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, $\boldsymbol{P}$, $\boldsymbol{B}$, 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 ($\boldsymbol{G}$), 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, bx,
                     CEED_VECTOR_NONE);
CeedOperatorSetField(op_setup, "dx", Erestrictx, bx, CEED_VECTOR_ACTIVE);
CeedOperatorSetField(op_setup, "rho", Erestrictui, CEED_BASIS_COLLOCATED,
                     CEED_VECTOR_ACTIVE);
```

```
CeedOperatorApply(op_setup, X, qdata, 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", Erestrictui,CEED_BASIS_COLLOCATED,
                     qdata);
CeedOperatorSetField(op_mass, "u", Erestrictu, bu, CEED_VECTOR_ACTIVE);
CeedOperatorSetField(op_mass, "v", Erestrictu, bu, 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).

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

int main(int argc, char **argv) {
  Ceed ceed;
  CeedVector in[16], out[16];
  CeedVector Qdata, J, W, U, V;
  CeedQFunction qf_setup, qf_mass;
  CeedInt Q = 8;
  const CeedScalar *vv;
  CeedScalar j[Q], w[Q], u[Q], v[Q];


  CeedInit(argv[1], &ceed);

  CeedQFunctionCreateInteriorByName(ceed, "Mass1DBuild", &qf_setup);
  CeedQFunctionCreateInteriorByName(ceed, "MassApply", &qf_mass);

  for (CeedInt i=0; i<Q; i++) {
    CeedScalar x = 2.*i/(Q-1) - 1;
    j[i] = 1;
    w[i] = 1 - x*x;
    u[i] = 2 + 3*x + 5*x*x;
    v[i] = w[i] * u[i];
  }

  CeedVectorCreate(ceed, Q, &J);
  CeedVectorSetArray(J, CEED_MEM_HOST, CEED_USE_POINTER, j);
  CeedVectorCreate(ceed, Q, &W);
  CeedVectorSetArray(W, CEED_MEM_HOST, CEED_USE_POINTER, w);
  CeedVectorCreate(ceed, Q, &U);
  CeedVectorSetArray(U, CEED_MEM_HOST, CEED_USE_POINTER, u);
  CeedVectorCreate(ceed, Q, &V);
  CeedVectorSetValue(V, 0);
  CeedVectorCreate(ceed, Q, &Qdata);
  CeedVectorSetValue(Qdata, 0);

  {
    in[0] = J;
    in[1] = W;
    out[0] = Qdata;
    CeedQFunctionApply(qf_setup, Q, in, out);
  }
  {
    in[0] = W;
    in[1] = U;
    out[0] = V;
    CeedQFunctionApply(qf_mass, Q, in, out);
```

(continues on next page)

(continued from previous page)

```
  }

  CeedVectorGetArrayRead(V, CEED_MEM_HOST, &vv);
  for (CeedInt i=0; i<Q; i++)
    if (v[i] != vv[i])
      // LCOV_EXCL_START
      printf("[%d] v %f != vv %f\n",i, v[i], vv[i]);
  // LCOV_EXCL_STOP
  CeedVectorRestoreArrayRead(V, &vv);

  CeedVectorDestroy(&W);
  CeedVectorDestroy(&U);
  CeedVectorDestroy(&V);
  CeedVectorDestroy(&Qdata);
  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 $\boldsymbol{x} = (x, y, z) \equiv (x_0, x_1, x_2) \in \Omega_e$, while the reference coordinates are represented as $\boldsymbol{X} = (X, Y, Z) \equiv (X_0, X_1, X_2) \in \mathrm{I} = [-1, 1]^3$ (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 PETSc demos and BPs

4.3.1 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 $\boldsymbol{X} = (X, Y) \equiv (X_0, X_1) \in \mathrm{I} = [-1, 1]^2$.

4.3.1.1 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

$$\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}}_{(2\times 2)} = \frac{\partial \boldsymbol{x}}{\partial \bar{\boldsymbol{x}}}_{(2\times 3)} \frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}}_{(3\times 2)}, \tag{4.4}$$

with Jacobian determinant given by

$$|J| = \left\| col_1 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right\| \left\| col_2 \left(\frac{\partial \bar{\boldsymbol{x}}}{\partial \boldsymbol{X}} \right) \right\| \tag{4.5}$$

We note that in equation (4.4), 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

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

4.3.1.2 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
```

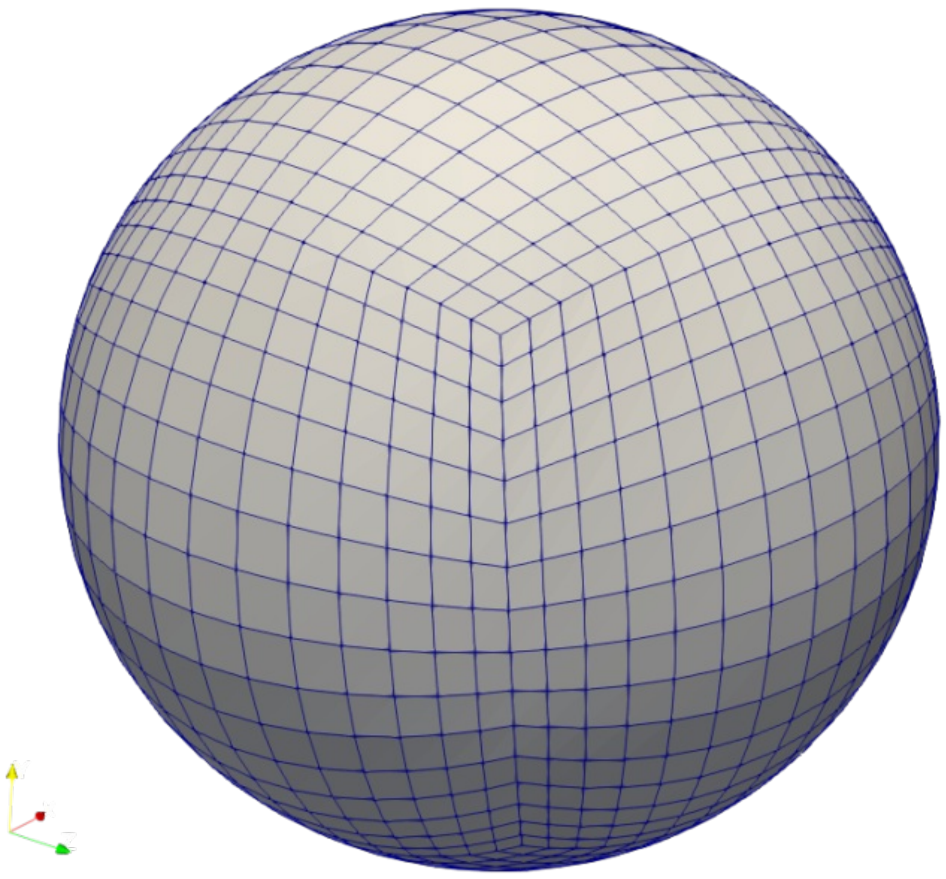


Fig. 4.1: Example of a cubed-sphere, i.e., a tensor-product discrete sphere, obtained by projecting a cube inscribed in a sphere onto the surface of the sphere.

This example uses the following coordinate transformations for the computation of the geometric factors: from the physical coordinates on the sphere, denoted by $\overset{\circ}{\boldsymbol{x}} = (\overset{\circ}{x}, \overset{\circ}{y}, \overset{\circ}{z})$, 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

$$\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}_{(3\times 2)} = \frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{x}}_{(3\times 3)} \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}}_{(3\times 2)}, \tag{4.6}$$

with Jacobian determinant given by

$$|J| = \left| col_1\left(\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}\right) \times col_2\left(\frac{\partial \mathring{\boldsymbol{x}}}{\partial \boldsymbol{X}}\right)\right|. \tag{4.7}$$

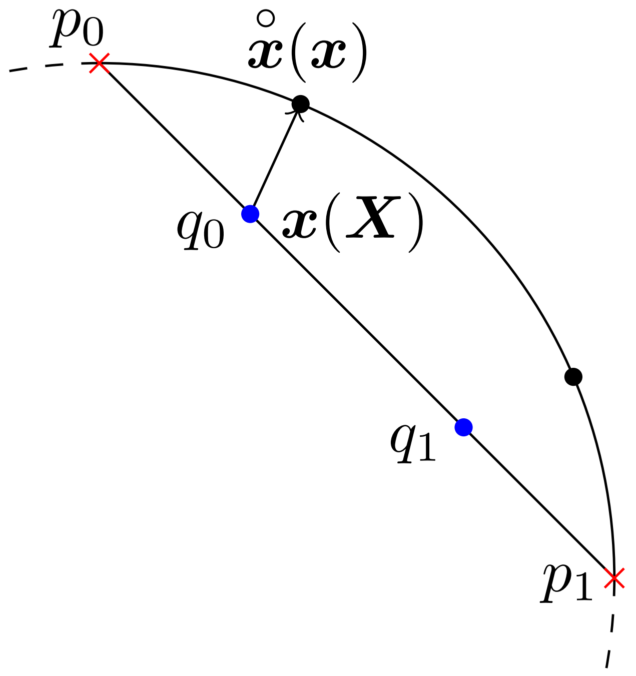

p_0
$\mathring{\boldsymbol{x}}(\boldsymbol{x})$
q_0
$\boldsymbol{x}(\boldsymbol{X})$
q_1
p_1

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 $\mathring{\boldsymbol{x}}(\boldsymbol{x})$.

We note that in equation (4.6), the right-most Jacobian matrix $\partial \boldsymbol{x}/\partial \boldsymbol{X}_{(3\times 2)}$ is provided by the library, while $\partial \mathring{\boldsymbol{x}}/\partial \boldsymbol{x}_{(3\times 3)}$ is provided by the user with analytical derivatives. In particular, for a sphere of radius 1, we have

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

and thus

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

4.3.2 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`.

4.3.2.1 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 (4.7), are the same as in the *Sphere* example. For the Poisson's problem, BP3-BP6, on the cubed-sphere, in addition to equation (4.7), the pseudo-inverse of $\partial \overset{\circ}{\boldsymbol{x}}/\partial \boldsymbol{X}$ 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:

$$\frac{\partial \boldsymbol{X}}{\partial \overset{\circ}{\boldsymbol{x}}}_{(2\times 3)} \equiv \left(\frac{\partial \overset{\circ}{\boldsymbol{x}}}{\partial \boldsymbol{X}}\right)^{+}_{(2\times 3)} = \left(\frac{\partial \overset{\circ}{\boldsymbol{x}}}{\partial \boldsymbol{X}}^T_{(2\times 3)} \frac{\partial \overset{\circ}{\boldsymbol{x}}}{\partial \boldsymbol{X}}_{(3\times 2)}\right)^{-1} \frac{\partial \overset{\circ}{\boldsymbol{x}}}{\partial \boldsymbol{X}}^T_{(2\times 3)}. \tag{4.8}$$

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

$$\frac{\partial u}{\partial \overset{\circ}{\boldsymbol{x}}}_{(1\times 3)} = \frac{\partial u}{\partial \boldsymbol{X}}_{(1\times 2)} \frac{\partial \boldsymbol{X}}{\partial \overset{\circ}{\boldsymbol{x}}}_{(2\times 3)}$$

and thus the weak Laplacian may be expressed as

$$\int_\Omega \frac{\partial v}{\partial \overset{\circ}{\boldsymbol{x}}} \left(\frac{\partial u}{\partial \overset{\circ}{\boldsymbol{x}}}\right)^T dS = \int_\Omega \frac{\partial v}{\partial \boldsymbol{X}} \underbrace{\frac{\partial \boldsymbol{X}}{\partial \overset{\circ}{\boldsymbol{x}}} \left(\frac{\partial \boldsymbol{X}}{\partial \overset{\circ}{\boldsymbol{x}}}\right)^T}_{\boldsymbol{g}_{(2\times 2)}} \left(\frac{\partial u}{\partial \boldsymbol{X}}\right)^T dS \tag{4.9}$$

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,

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

where we have dropped the transpose due to symmetry. This allows us to simplify (4.9) as

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

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

4.3.3 Multigrid

This example is located in the subdirectory `examples/petsc`. It investigates p-multigrid for the Poisson problem, equation (4.10), using an unstructured high-order finite element discretization. All of the operators associated with the geometric multigrid are implemented in libCEED.

$$-\nabla \cdot (\kappa(x) \nabla x) = g(x) \tag{4.10}$$

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.

4.4 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	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.

4.4.1 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 $v \in V^p$

$$\langle v, u \rangle = \langle v, f \rangle, \tag{4.11}$$

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

$$\begin{aligned} u(\boldsymbol{x}) &= \sum_{j=1}^{n} u_j \, \phi_j(\boldsymbol{x}), \\ v(\boldsymbol{x}) &= \sum_{i=1}^{n} v_i \, \phi_i(\boldsymbol{x}). \end{aligned} \tag{4.12}$$

The coefficients $\{u_j\}$ and $\{v_i\}$ are the nodal values of u and v, respectively. Inserting the expressions (4.12) into (4.11), we obtain the inner-products

$$\langle v, u \rangle = \boldsymbol{v}^T M \boldsymbol{u}, \qquad \langle v, f \rangle = \boldsymbol{v}^T \boldsymbol{b} \,. \tag{4.13}$$

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 $i, j \in \{1, \ldots, n\}$.

4.4.2 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 (4.12), 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 $i, j \in \{1, \ldots, n\}$.

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.

The mathematical formulation (from [GRL10], cf. SE3) is given in what follows. The compressible Navier-Stokes equations in conservative form are

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

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), $\boldsymbol{I}_3$ represents the 3×3 identity matrix, g the gravitational acceleration constant, $\hat{\boldsymbol{k}}$ the unit vector in the z direction, k the thermal conductivity constant, T represents the temperature, and P the pressure, given by the following equation of state

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

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

$$\begin{aligned} \boldsymbol{F}(\boldsymbol{q}) &= \begin{pmatrix} \boldsymbol{U} \\ (\boldsymbol{U} \otimes \boldsymbol{U})/\rho + P \boldsymbol{I}_3 - \boldsymbol{\sigma} \\ (E + P)\boldsymbol{U}/\rho - \boldsymbol{u} \cdot \boldsymbol{\sigma} - k \nabla T \end{pmatrix} , \\ S(\boldsymbol{q}) &= - \begin{pmatrix} 0 \\ \rho g \hat{\boldsymbol{k}} \\ 0 \end{pmatrix} . \end{aligned}$$

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 $\psi_{kji} = h_i(X_0)h_j(X_1)h_k(X_2)$.

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

- 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 $\boldsymbol{q}_N$:

$$\boldsymbol{f}(\boldsymbol{q}_N) \equiv \boldsymbol{g}(t^{n+1}, \boldsymbol{q}_N, \dot{\boldsymbol{q}}_N) = 0 \,, \tag{4.17}$$

  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 (4.17) will correspond to a weak form, as explained below. In determining how difficult a given problem is to solve, we consider the Jacobian of (4.17),

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

  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.

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,

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

where $\boldsymbol{F}(\boldsymbol{q}_N) \cdot \widehat{\boldsymbol{n}}$ 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 $\boldsymbol{F} \cdot \widehat{\boldsymbol{n}}$ contracts over spatial dimensions.

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

$$\begin{aligned}\int_\Omega \boldsymbol{v} \cdot \left(\frac{\partial \boldsymbol{q}_N}{\partial t} - \boldsymbol{S}(\boldsymbol{q}_N)\right) dV - \int_\Omega \nabla \boldsymbol{v} \!:\! \boldsymbol{F}(\boldsymbol{q}_N)\, dV &\\ + \int_{\partial\Omega} \boldsymbol{v} \cdot \boldsymbol{F}(\boldsymbol{q}_N) \cdot \widehat{\boldsymbol{n}}\, dS &\\ + \int_\Omega \boldsymbol{P}(\boldsymbol{v})^T \left(\frac{\partial \boldsymbol{q}_N}{\partial t} + \nabla \cdot \boldsymbol{F}(\boldsymbol{q}_N) - \boldsymbol{S}(\boldsymbol{q}_N)\right) dV = 0\,,\ \forall \boldsymbol{v} \in \mathcal{V}_p&\end{aligned} \tag{4.19}$$

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

- **SU** (streamline-upwind)

  This method is a simplified version of *SUPG* (4.19) which is developed for debugging/comparison purposes. The weak form for this method is

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

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

In both (4.20) and (4.19), $\boldsymbol{P}$ is called the *perturbation to the test-function space*, since it modifies the original Galerkin method into *SUPG* or *SU* schemes. It is defined as

$$\boldsymbol{P}(\boldsymbol{v}) \equiv \left(\boldsymbol{\tau} \cdot \frac{\partial \boldsymbol{F}(\boldsymbol{q}_N)}{\partial \boldsymbol{q}_N}\right)^T \nabla \boldsymbol{v} \, ,$$

where parameter $\boldsymbol{\tau} \in \mathbb{R}^{3\times 3}$ is an intrinsic time/space scale matrix.

Currently, this demo provides two types of problems/physical models that can be selected at run time via the option `-problem`. One is the problem of transport of energy in a uniform vector velocity field, called the *Advection* problem, and is the so called *Density Current* problem.

4.5.1 Advection

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

$$\frac{\partial E}{\partial t} + \nabla \cdot (\boldsymbol{u} E) = 0 \, , \tag{4.21}$$

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 (4.21) applying zero energy density E, and no-flux for $\boldsymbol{u}$ on the boundaries.

  The $3D$ version of this test case can be run with:

```
./navierstokes -problem advection -problem_advection_wind rotation
```

  while the $2D$ version with:

```
./navierstokes -problem advection2d -problem_advection_wind rotation
```

- **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

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

  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

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

  The $3D$ version of this test case problem can be run with:

```
./navierstokes -problem advection -problem_advection_wind translation -
↪problem_advection_wind translation .5,-1,0
```

while the $2D$ version with:

```
./navierstokes -problem advection2d -problem_advection_wind translation -
↪problem_advection_wind translation 1,-.5
```

4.5.2 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

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

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. This problem can be run with:

```
./navierstokes -problem density_current
```

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 $\boldsymbol{\sigma}$ and $\boldsymbol{\epsilon}$ 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 reference configuration, and C is a linearized constitutive model.

$$
\begin{tikzcd}
\overbrace{\boldsymbol{S}(\boldsymbol{E})}^{\text{Finite Strain Hyperelastic}} \arrow[r, "\text{constitutive}", "\text{linearization}"'] \arrow[d, "\substack{\boldsymbol{E}\to\boldsymbol{\epsilon} \\ \boldsymbol{S}\to\boldsymbol{\sigma}}", "\text{geometric}"'] & \overbrace{\boldsymbol{S} = \mathsf{C}\boldsymbol{E}}^{\text{St. Venant-Kirchoff}} \arrow[d, "\substack{\boldsymbol{E}\to\boldsymbol{\epsilon} \\ \boldsymbol{S}\to\boldsymbol{\sigma}}"', "\text{geometric}"] \\
\underbrace{\boldsymbol{\sigma}(\boldsymbol{\epsilon})}_{\text{Small Strain Hyperelastic}} \arrow[r, "\text{constitutive}", "\text{linearization}"'] & \underbrace{\boldsymbol{\sigma} = \mathsf{C}\boldsymbol{\epsilon}}_{\text{Linear Elastic}}
\end{tikzcd}
\tag{4.22}
$$

4.6.1 Running the mini-app

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

Table 4.1: Mandatory Runtime Options

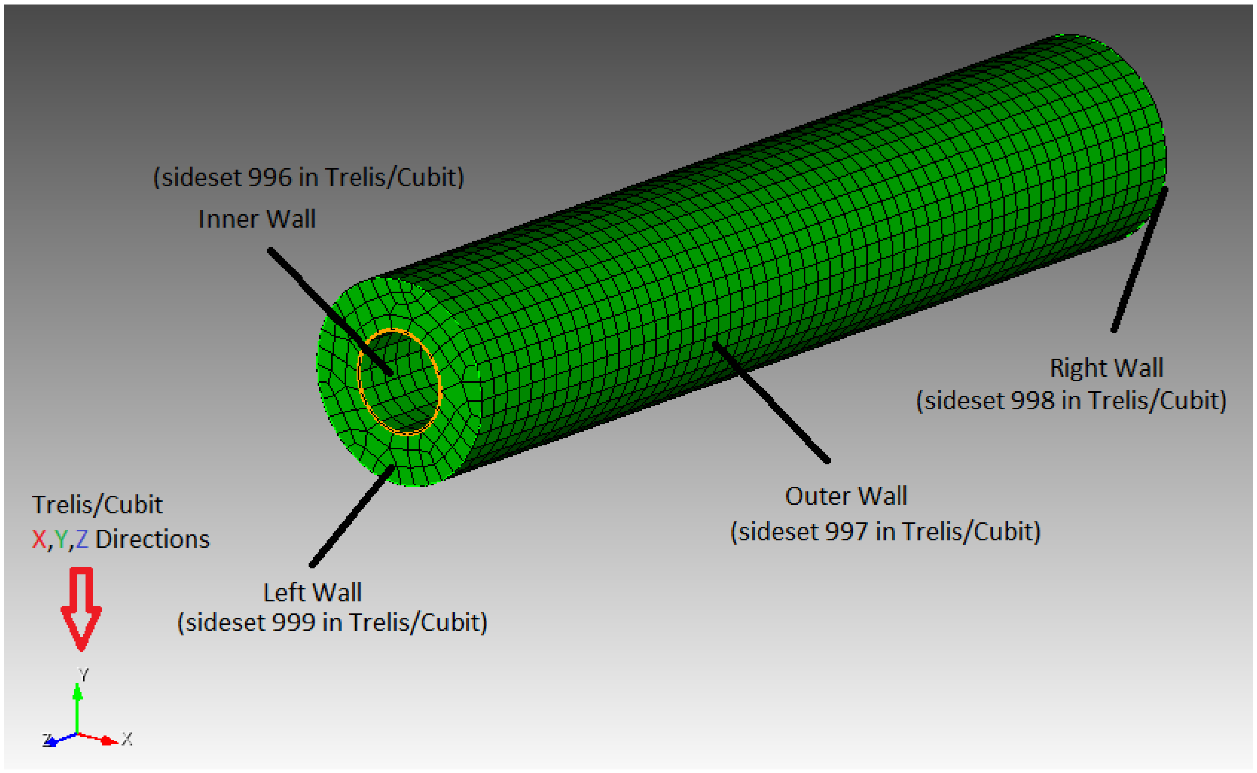
Option	Description
`-mesh [filename]`	Path to mesh file in any format supported by PETSc.
`-degree [int]`	Polynomial degree of the finite element basis
`-E [real]`	Young's modulus, $E > 0$
`-nu [real]`	Poisson's ratio, $\nu < 0.5$
`-bc_clamp [int list]`	List of face sets on which to displace by `-bc_clamp_[facenumber]_translate [x,y,z]` and/or `bc_clamp_[facenumber]_rotate [rx,ry,rz,theta]`

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)
Outer Wall
(sideset 997 in Trelis/Cubit)
Trelis/Cubit
X,Y,Z Directions
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]`.

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 4.2: Additional Runtime Options

Option	Description	Default value
`-ceed`	CEED resource specifier	`/cpu/self`
`-qextra`	Number of extra quadrature points	`0`
`-test`	Run in test mode	
`-problem`	Problem to solve (`linElas`, `hyperSS` or `hyperFS`)	`linElas`
`-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 `linElas` 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	
`-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 4.3: (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 4.4: 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, $\int \rho \boldsymbol{g}$	$5 \cdot 10^{4}\,\mathrm{kg\,m^{-2}\,s^{-2}} \cdot (\mathrm{volume}\,\mathrm{m}^3)$

One can choose units of displacement independently (e.g., `-units_meter 100` to measure displacement in centimeters), but E and $\int \rho \boldsymbol{g}$ 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, $\operatorname{trace} \boldsymbol{E}$, $\operatorname{trace} \boldsymbol{E}^2$, $|J|$, and strain energy density. The table below summarizes the formulations of each of these quantities for each problem type.

Table 4.5: Diagnostic quantities

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

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]:

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{g} = \boldsymbol{0} \tag{4.23}$$

where $\boldsymbol{\sigma}$ and $\boldsymbol{g}$ are stress and forcing functions, respectively. We multiply (4.23) 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

$$\int_{\Omega} \nabla \boldsymbol{v} \colon \boldsymbol{\sigma} \, dV - \int_{\partial \Omega} \boldsymbol{v} \cdot (\boldsymbol{\sigma} \cdot \hat{\boldsymbol{n}}) \, dS - \int_{\Omega} \boldsymbol{v} \cdot \boldsymbol{g} \, dV = 0, \quad \forall \boldsymbol{v} \in \mathcal{V}, \tag{4.24}$$

where $\boldsymbol{\sigma} \cdot \hat{\boldsymbol{n}}|_{\partial \Omega}$ is replaced by an applied force/traction boundary condition written in terms of the reference configuration.

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

$$\boldsymbol{\epsilon} = \frac{1}{2} \left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \right). \tag{4.25}$$

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

$$Q \boldsymbol{\sigma}(\boldsymbol{\epsilon}) Q^T = \boldsymbol{\sigma}(Q \boldsymbol{\epsilon} Q^T), \tag{4.26}$$

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 (4.26) 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,

$$\boldsymbol{\sigma}(\boldsymbol{\epsilon}) = \frac{\partial \Phi}{\partial \boldsymbol{\epsilon}}. \tag{4.27}$$

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

$$\boldsymbol{\sigma} = \mathsf{C} : \boldsymbol{\epsilon}. \tag{4.28}$$

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

$$\mathsf{C} = \begin{pmatrix} \lambda + 2\mu & \lambda & \lambda & & & \\ \lambda & \lambda + 2\mu & \lambda & & & \\ \lambda & \lambda & \lambda + 2\mu & & & \\ & & & \mu & & \\ & & & & \mu & \\ & & & & & \mu \end{pmatrix}. \tag{4.29}$$

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 (4.23) and (4.24), 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

$$\boldsymbol{\sigma} = \lambda \log(1 + \operatorname{trace}\boldsymbol{\epsilon})\boldsymbol{I}_3 + 2\mu\boldsymbol{\epsilon} \tag{4.30}$$

where $\boldsymbol{\epsilon}$ is defined as in (4.25).

4.6.3.1 Newton linearization

Due to nonlinearity in the constitutive law, we require a Newton linearization of (4.30). 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,

$$\mathrm{d}\boldsymbol{\sigma} = \bar{\lambda} \cdot \operatorname{trace} \mathrm{d}\boldsymbol{\epsilon} \cdot \boldsymbol{I}_3 + 2\mu \, \mathrm{d}\boldsymbol{\epsilon} \tag{4.31}$$

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

$$\begin{pmatrix} \mathrm{d}\sigma_{11} \\ \mathrm{d}\sigma_{22} \\ \mathrm{d}\sigma_{33} \\ \mathrm{d}\sigma_{23} \\ \mathrm{d}\sigma_{13} \\ \mathrm{d}\sigma_{12} \end{pmatrix} = \begin{pmatrix} 2\mu + \bar{\lambda} & \bar{\lambda} & \bar{\lambda} & & & \\ \bar{\lambda} & 2\mu + \bar{\lambda} & \bar{\lambda} & & & \\ \bar{\lambda} & \bar{\lambda} & 2\mu + \bar{\lambda} & & & \\ & & & \mu & & \\ & & & & \mu & \\ & & & & & \mu \end{pmatrix} \begin{pmatrix} \mathrm{d}\epsilon_{11} \\ \mathrm{d}\epsilon_{22} \\ \mathrm{d}\epsilon_{33} \\ 2\,\mathrm{d}\epsilon_{23} \\ 2\,\mathrm{d}\epsilon_{13} \\ 2\,\mathrm{d}\epsilon_{12} \end{pmatrix}. \tag{4.32}$$

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 reference 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 reference 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:

$$-\nabla_X \cdot \boldsymbol{P} - \rho_0 \boldsymbol{g} = \boldsymbol{0} \tag{4.33}$$

where the $_X$ in ∇_X indicates that the gradient is calculated with respect to the reference 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 *reference* mass density. The tensor $\boldsymbol{P}$ is not symmetric, living in the current configuration on the left and the reference configuration on the right.

$\boldsymbol{P}$ can be decomposed as

$$\boldsymbol{P} = \boldsymbol{F}\,\boldsymbol{S}, \tag{4.34}$$

where $\boldsymbol{S}$ is the *second Piola-Kirchhoff stress* tensor, a symmetric tensor defined entirely in the reference 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 reference configuration, the right Cauchy-Green tensor

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

and the Green-Lagrange strain tensor

$$\boldsymbol{E} = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I}_3) = \frac{1}{2}\Big(\nabla_X \boldsymbol{u} + (\nabla_X \boldsymbol{u})^T + (\nabla_X \boldsymbol{u})^T \nabla_X \boldsymbol{u}\Big), \tag{4.35}$$

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 (4.28), 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,

$$\begin{aligned} \Phi(\boldsymbol{E}) &= \frac{\lambda}{2}(\log J)^2 + \frac{\mu}{2}(\operatorname{trace}\boldsymbol{C} - 3) - \mu \log J \\ &= \frac{\lambda}{2}(\log J)^2 + \mu \operatorname{trace}\boldsymbol{E} - \mu \log J, \end{aligned} \tag{4.36}$$

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 (4.27), 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 (4.27) for the model (4.36), we arrive at

$$\boldsymbol{S} = \lambda \log J\boldsymbol{C}^{-1} + \mu(\boldsymbol{I}_3 - \boldsymbol{C}^{-1}). \tag{4.37}$$

Tip: An equivalent form of (4.37) is

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

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 (4.35).

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 symmetric matrix $C = \begin{pmatrix} 1+e_{00} & e_{01} \\ e_{01} & 1+e_{11} \end{pmatrix}$. Then we compute

$$\log \sqrt{|C|} = \frac{1}{2}\texttt{log1p}(e_{00} + e_{11} + e_{00}e_{11} - e_{01}^2).$$

which gives accurate results even in the limit when the entries e_{ij} are very small. For example, if $e_{ij} \sim 10^{-8}$, 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.

Note: One can linearize (4.37) 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 (4.37) reduces to

$$\boldsymbol{S} = \lambda(\operatorname{trace}\boldsymbol{E})\boldsymbol{I}_3 + 2\mu\boldsymbol{E}, \tag{4.38}$$

which is the St. Venant-Kirchoff model (constitutive linearization without geometric linearization; see (4.22)).

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 (4.30) (see (4.22)).

4.6.4.2 Weak form

We multiply (4.33) 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

$$\int_{\Omega_0} \nabla_X \boldsymbol{v} \colon \boldsymbol{P} \, dV - \int_{\Omega_0} \boldsymbol{v} \cdot \rho_0 \boldsymbol{g} \, dV - \int_{\partial \Omega_0} \boldsymbol{v} \cdot (\boldsymbol{P} \cdot \hat{\boldsymbol{N}}) \, dS = 0, \quad \forall \boldsymbol{v} \in \mathcal{V}, \tag{4.39}$$

where $\boldsymbol{P} \cdot \hat{\boldsymbol{N}}|_{\partial \Omega}$ is replaced by any prescribed force/traction boundary condition written in terms of the reference 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 reference configuration. Discretization of (4.39) 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 (4.39), we begin by expressing the derivative of (4.34) in incremental form,

$$\mathrm{d}\boldsymbol{P} = \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}} \colon \mathrm{d}\boldsymbol{F} = \mathrm{d}\boldsymbol{F} \, \boldsymbol{S} + \boldsymbol{F} \underbrace{\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}} \colon \mathrm{d}\boldsymbol{E}}_{\mathrm{d}\boldsymbol{S}} \tag{4.40}$$

where

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

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 (4.29). We now evaluate $\mathrm{d}\boldsymbol{S}$ for the Neo-Hookean model (4.37),

$$\mathrm{d}\boldsymbol{S} = \frac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}} \colon \mathrm{d}\boldsymbol{E} = \lambda (\boldsymbol{C}^{-1} \colon \mathrm{d}\boldsymbol{E}) \boldsymbol{C}^{-1} + 2(\mu - \lambda \log J) \boldsymbol{C}^{-1} \, \mathrm{d}\boldsymbol{E} \, \boldsymbol{C}^{-1}, \tag{4.41}$$

where we have used

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

Note: In the small-strain limit, $\boldsymbol{C} \to \boldsymbol{I}_3$ and $\log J \to 0$, thereby reducing (4.41) to the St. Venant-Kirchoff model (4.38).

Note: Some cancellation is possible (at the expense of symmetry) if we substitute (4.41) into (4.40),

$$\begin{aligned}
\mathrm{d}\boldsymbol{P} &= \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \lambda(\boldsymbol{C}^{-1} : \mathrm{d}\boldsymbol{E})\boldsymbol{F}^{-T} + 2(\mu - \lambda\log J)\boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{E}\,\boldsymbol{C}^{-1} \\
&= \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \lambda(\boldsymbol{F}^{-T} : \mathrm{d}\boldsymbol{F})\boldsymbol{F}^{-T} + (\mu - \lambda\log J)\boldsymbol{F}^{-T}(\boldsymbol{F}^{T}\,\mathrm{d}\boldsymbol{F} + \mathrm{d}\boldsymbol{F}^{T}\boldsymbol{F})\boldsymbol{C}^{-1} \\
&= \mathrm{d}\boldsymbol{F}\,\boldsymbol{S} + \lambda(\boldsymbol{F}^{-T} : \mathrm{d}\boldsymbol{F})\boldsymbol{F}^{-T} + (\mu - \lambda\log J)\Big(\mathrm{d}\boldsymbol{F}\,\boldsymbol{C}^{-1} + \boldsymbol{F}^{-T}\,\mathrm{d}\boldsymbol{F}^{T}\boldsymbol{F}^{-T}\Big),
\end{aligned} \tag{4.42}$$

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

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

We prefer to compute with (4.41) because (4.42) 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.

It is sometimes useful to express (4.41) in index notation,

$$\begin{aligned}
\mathrm{d}\boldsymbol{S}_{IJ} &= \frac{\partial \boldsymbol{S}_{IJ}}{\partial \boldsymbol{E}_{KL}}\,\mathrm{d}\boldsymbol{E}_{KL} \\
&= \lambda(\boldsymbol{C}^{-1}_{KL}\,\mathrm{d}\boldsymbol{E}_{KL})\boldsymbol{C}^{-1}_{IJ} + 2(\mu - \lambda\log J)\boldsymbol{C}^{-1}_{IK}\,\mathrm{d}\boldsymbol{E}_{KL}\boldsymbol{C}^{-1}_{LJ} \\
&= \underbrace{\Big(\lambda\boldsymbol{C}^{-1}_{IJ}\boldsymbol{C}^{-1}_{KL} + 2(\mu - \lambda\log J)\boldsymbol{C}^{-1}_{IK}\boldsymbol{C}^{-1}_{JL}\Big)}_{\mathsf{C}_{IJKL}}\mathrm{d}\boldsymbol{E}_{KL}\,,
\end{aligned} \tag{4.43}$$

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 (4.41) or the second line of (4.43) to compute $\mathrm{d}\boldsymbol{S}$ while avoiding computation or storage of higher order tensors, and
3. conclude by (4.40), where $\boldsymbol{S}$ is either stored or recomputed from its definition exactly as in the nonlinear residual evaluation.

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 (4.43), 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.

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`**

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 *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
CeedWait(&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 *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

Return An error code: 0 - success, otherwise - failure

Parameters

- `req`: Address of CeedRequest to wait for; zeroed on completion.

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

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

User Functions

See *CeedRegister() CeedDestroy()*

Return An error code: 0 - success, otherwise - failure

Parameters

- `resource`: Resource to use, e.g., "/cpu/self"
- `ceed`: The library context

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

Get the full resource name for a Ceed context.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to get resource name of
- `[out] resource`: Variable to store resource name

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

Return Ceed context preferred memory type.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to get preferred memory type of
- `[out] type`: Address to save preferred memory type to

int **`CeedIsDeterministic`**(*Ceed ceed*, bool **isDeterministic*)

Get deterministic status of Ceed.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] ceed`: Ceed
- `[out] isDeterministic`: Variable to store deterministic status

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

View a Ceed.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] ceed`: Ceed to view
- `[in] stream`: Filestream to write to

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

Destroy a Ceed context.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Address of Ceed context to destroy

`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 *lineno*, **`const`** char **func*, int *ecode*, **`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 *lineno*, **`const`** char **func*, int *ecode*, **`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 *lineno*, **const** char **func*, int *ecode*, **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 *lineno*, **const** char **func*, int *ecode*, **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*, int (**eh*))*Ceed*, **const** char*, int, **const** char*, int, **const** char*, va_list*

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 ***errmsg*)

Get error message.

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

Library Developer Functions

Parameters

- `[in] ceed`: Ceed contex to retrieve error message
- `[out] errmsg`: Char pointer to hold error message

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

Restore error message.

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

Library Developer Functions

Parameters

- `[in] ceed`: Ceed contex to restore error message
- `[out] errmsg`: Char pointer that holds error message

Macros

`CeedError`(*ceed*, *ecode*, *...*)
: Raise an error on ceed object.

  See *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.

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)

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_WEIGHTS).

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

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

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

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

int **`CeedVectorSetArray`**(*CeedVector vec*, *CeedMemType mtype*, *CeedCopyMode cmode*, *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

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector
- `mtype`: Memory type of the array being passed
- `cmode`: Copy mode for the array
- `array`: Array to be used, or NULL with *CEED_COPY_VALUES* to have the library allocate

int **`CeedVectorSetValue`**(*CeedVector vec*, *CeedScalar value*)

Set the CeedVector to a constant value.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector
- `[in] value`: Value to be used

int **`CeedVectorSyncArray`**(*CeedVector vec*, *CeedMemType mtype*)

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

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector
- `mtype`: Memtype to be synced

int **`CeedVectorTakeArray`**(*CeedVector vec*, *CeedMemType mtype*, *CeedScalar **array*)

Take ownership of the CeedVector array and remove the array from the CeedVector.

The caller is responsible for managing and freeing the array.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector
- `mtype`: Memory type on which to take the array. If the backend uses a different memory type, this will perform a copy.
- `[out] array`: Array on memory type mtype, or NULL if array pointer is not required

int **`CeedVectorGetArray`**(*CeedVector vec*, *CeedMemType mtype*, *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.

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to access
- `mtype`: Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy.
- `[out] array`: Array on memory type mtype

int **`CeedVectorGetArrayRead`**(*CeedVector vec*, *CeedMemType mtype*, **`const`** *CeedScalar **array*)

Get read-only access to a CeedVector via the specified memory type.

Restore access with *CeedVectorRestoreArrayRead()*.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to access
- `mtype`: Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy (possibly cached).
- `[out] array`: Array on memory type mtype

int **`CeedVectorRestoreArray`**(*CeedVector vec*, *CeedScalar **array*)

Restore an array obtained using *CeedVectorGetArray()*

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to restore
- `array`: Array of vector data

int **`CeedVectorRestoreArrayRead`**(*CeedVector vec*, **`const`** *CeedScalar **array*)

Restore an array obtained using *CeedVectorGetArrayRead()*

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to restore
- `array`: Array of vector data

int **`CeedVectorNorm`**(*CeedVector vec*, *CeedNormType 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

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to retrieve maximum value
- `type`: Norm type *CEED_NORM_1*, *CEED_NORM_2*, or *CEED_NORM_MAX*
- `[out] norm`: Variable to store norm value

int **`CeedVectorReciprocal`**(*CeedVector vec*)

Take the reciprocal of a CeedVector.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to take reciprocal

int **`CeedVectorView`**(*CeedVector vec*, **const** char **fpfmt*, FILE **stream*)

View a CeedVector.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] vec`: CeedVector to view
- `[in] fpfmt`: Printing format
- `[in] stream`: Filestream to write to

int **`CeedVectorGetLength`**(*CeedVector vec*, *CeedInt *length*)

Get the length of a CeedVector.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to retrieve length
- `[out] length`: Variable to store length

int **`CeedVectorDestroy`**(*CeedVector *vec*)

Destroy a CeedVector.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to destroy

Typedefs and Enumerations

`typedef` int32_t **`CeedInt`**
Integer type, used for indexing.

`typedef` double **`CeedScalar`**
Scalar (floating point) type.

`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(3).

`enum CeedNormType`
Denotes type of vector norm to be computed.

Values:

`enumerator CEED_NORM_1`
L_1 norm: sum_i |x_i|.

`enumerator CEED_NORM_2`
L_2 norm: sqrt(sum_i |x_i|^2)

`enumerator CEED_NORM_MAX`
L_Infinity norm: 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] = {}

Indicate that the stride is determined by the backend.

const *CeedElemRestriction* **CEED_ELEMRESTRICTION_NONE** = &ceed_elemrestriction_none

Indicate that no CeedElemRestriction is provided by the user.

int **CeedElemRestrictionCreate**(*Ceed ceed*, *CeedInt nelem*, *CeedInt elemsize*, *CeedInt ncomp*, *CeedInt compstride*, *CeedInt lsize*, *CeedMemType mtype*, *CeedCopyMode cmode*, **const** *CeedInt *offsets*, *CeedElemRestriction *rstr*)

Create a CeedElemRestriction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedElemRestriction will be created
- `nelem`: Number of elements described in the *offsets* array
- `elemsize`: Size (number of "nodes") per element
- `ncomp`: Number of field components per interpolation node (1 for scalar fields)
- `compstride`: 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*elemsize] + j*compstride.
- `lsize`: The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- `mtype`: Memory type of the *offsets* array, see CeedMemType
- `cmode`: Copy mode for the *offsets* array, see CeedCopyMode
- `offsets`: Array of shape [*nelem*, *elemsize*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *nelem*. All offsets must be in the range [0, *lsize* - 1].
- `[out] rstr`: Address of the variable where the newly created CeedElemRestriction will be stored

int **CeedElemRestrictionCreateStrided**(*Ceed ceed*, *CeedInt nelem*, *CeedInt elemsize*, *CeedInt ncomp*, *CeedInt lsize*, **const** *CeedInt strides*[3], *CeedElemRestriction *rstr*)

Create a strided CeedElemRestriction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedElemRestriction will be created
- `nelem`: Number of elements described by the restriction
- `elemsize`: Size (number of "nodes") per element
- `ncomp`: Number of field components per interpolation "node" (1 for scalar fields)

- `lsize`: The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- `strides`: 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`: Address of the variable where the newly created CeedElemRestriction will be stored

int **CeedElemRestrictionCreateBlocked**(*Ceed ceed*, *CeedInt nelem*, *CeedInt elemsize*, *CeedInt blksize*, *CeedInt ncomp*, *CeedInt compstride*, *CeedInt lsize*, *CeedMemType mtype*, *CeedCopyMode cmode*, **const** *CeedInt *offsets*, *CeedElemRestriction *rstr*)

Create a blocked CeedElemRestriction, typically only called by backends.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedElemRestriction will be created.
- `nelem`: Number of elements described in the *offsets* array.
- `elemsize`: Size (number of unknowns) per element
- `blksize`: Number of elements in a block
- `ncomp`: Number of field components per interpolation node (1 for scalar fields)
- `compstride`: 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*elemsize] + j*compstride.
- `lsize`: The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- `mtype`: Memory type of the *offsets* array, see CeedMemType
- `cmode`: Copy mode for the *offsets* array, see CeedCopyMode
- `offsets`: Array of shape [*nelem*, *elemsize*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *nelem*. All offsets must be in the range [0, *lsize* - 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`: Address of the variable where the newly created CeedElemRestriction will be stored

int **CeedElemRestrictionCreateBlockedStrided**(*Ceed ceed*, *CeedInt nelem*, *CeedInt elemsize*, *CeedInt blksize*, *CeedInt ncomp*, *CeedInt lsize*, **const** *CeedInt strides*[3], *CeedElemRestriction *rstr*)

Create a blocked strided CeedElemRestriction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedElemRestriction will be created
- `nelem`: Number of elements described by the restriction

- `elemsize`: Size (number of "nodes") per element
- `blksize`: Number of elements in a block
- `ncomp`: Number of field components per interpolation node (1 for scalar fields)
- `lsize`: The size of the L-vector. This vector may be larger than the elements and fields given by this restriction.
- `strides`: 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`: Address of the variable where the newly created CeedElemRestriction will be stored

int **`CeedElemRestrictionCreateVector`**(*CeedElemRestriction rstr*, *CeedVector *lvec*, *CeedVector *evec*)

Create CeedVectors associated with a CeedElemRestriction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `lvec`: The address of the L-vector to be created, or NULL
- `evec`: The address of the E-vector to be created, or NULL

int **`CeedElemRestrictionApply`**(*CeedElemRestriction rstr*, *CeedTransposeMode tmode*, *CeedVector u*, *CeedVector ru*, *CeedRequest *request*)

Restrict an L-vector to an E-vector or apply its transpose.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `tmode`: Apply restriction or transpose
- `u`: Input vector (of size *lsize* when tmode=*CEED_NOTRANSPOSE*)
- `ru`: Output vector (of shape [*nelem * elemsize*] when tmode=*CEED_NOTRANSPOSE*). Ordering of the e-vector is decided by the backend.
- `request`: Request or *CEED_REQUEST_IMMEDIATE*

int **`CeedElemRestrictionApplyBlock`**(*CeedElemRestriction rstr*, *CeedInt block*, *CeedTransposeMode tmode*, *CeedVector u*, *CeedVector ru*, *CeedRequest *request*)

Restrict an L-vector to a block of an E-vector or apply its transpose.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `block`: Block number to restrict to/from, i.e. block=0 will handle elements [0 : blksize] and block=3 will handle elements [3*blksize : 4*blksize]
- `tmode`: Apply restriction or transpose

- `u`: Input vector (of size *lsize* when tmode=*CEED_NOTRANSPOSE*)
- `ru`: Output vector (of shape [*blksize* * *elemsize*] when tmode=*CEED_NOTRANSPOSE*). Ordering of the e-vector is decided by the backend.
- `request`: Request or *CEED_REQUEST_IMMEDIATE*

int **`CeedElemRestrictionGetCompStride`**(*CeedElemRestriction rstr*, *CeedInt *compstride*)

Get the L-vector component stride.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] compstride`: Variable to store component stride

int **`CeedElemRestrictionGetNumElements`**(*CeedElemRestriction rstr*, *CeedInt *numelem*)

Get the total number of elements in the range of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] numelem`: Variable to store number of elements

int **`CeedElemRestrictionGetElementSize`**(*CeedElemRestriction rstr*, *CeedInt *elemsize*)

Get the size of elements in the CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] elemsize`: Variable to store size of elements

int **`CeedElemRestrictionGetLVectorSize`**(*CeedElemRestriction rstr*, *CeedInt *lsize*)

Get the size of the l-vector for a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] numnodes`: Variable to store number of nodes

int **`CeedElemRestrictionGetNumComponents`**(*CeedElemRestriction rstr*, *CeedInt *numcomp*)

Get the number of components in the elements of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction

- `[out] numcomp`: Variable to store number of components

int **`CeedElemRestrictionGetNumBlocks`**(*CeedElemRestriction rstr*, *CeedInt *numblock*)

Get the number of blocks in a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] numblock`: Variable to store number of blocks

int **`CeedElemRestrictionGetBlockSize`**(*CeedElemRestriction rstr*, *CeedInt *blksize*)

Get the size of blocks in the CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] blksize`: Variable to store size of blocks

int **`CeedElemRestrictionGetMultiplicity`**(*CeedElemRestriction rstr*, *CeedVector mult*)

Get the multiplicity of nodes in a CeedElemRestriction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] mult`: Vector to store multiplicity (of size lsize)

int **`CeedElemRestrictionView`**(*CeedElemRestriction rstr*, FILE **stream*)

View a CeedElemRestriction.

User Functions

Return Error code: 0 - success, otherwise - failure

Parameters

- `[in] rstr`: CeedElemRestriction to view
- `[in] stream`: Stream to write; typically stdout/stderr or a file

int **`CeedElemRestrictionDestroy`**(*CeedElemRestriction *rstr*)

Destroy a CeedElemRestriction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction to destroy

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_COLLOCATED** = &ceed_basis_collocated

Indicate that the quadrature points are collocated with the nodes.

int **CeedBasisCreateTensorH1**(*Ceed ceed*, *CeedInt dim*, *CeedInt ncomp*, *CeedInt P1d*, *CeedInt Q1d*, **const** *CeedScalar *interp1d*, **const** *CeedScalar *grad1d*, **const** *CeedScalar *qref1d*, **const** *CeedScalar *qweight1d*, *CeedBasis *basis*)

Create a tensor-product basis for H^1 discretizations.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedBasis will be created
- `dim`: Topological dimension
- `ncomp`: Number of field components (1 for scalar fields)
- `P1d`: Number of nodes in one dimension
- `Q1d`: Number of quadrature points in one dimension
- `interp1d`: Row-major (Q1d * P1d) matrix expressing the values of nodal basis functions at quadrature points
- `grad1d`: Row-major (Q1d * P1d) matrix expressing derivatives of nodal basis functions at quadrature points
- `qref1d`: Array of length Q1d holding the locations of quadrature points on the 1D reference element [-1, 1]
- `qweight1d`: Array of length Q1d holding the quadrature weights on the reference element
- `[out] basis`: Address of the variable where the newly created CeedBasis will be stored.

int **CeedBasisCreateTensorH1Lagrange**(*Ceed ceed*, *CeedInt dim*, *CeedInt ncomp*, *CeedInt P*, *CeedInt Q*, *CeedQuadMode qmode*, *CeedBasis *basis*)

Create a tensor-product Lagrange basis.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedBasis will be created
- `dim`: Topological dimension of element
- `ncomp`: Number of field components (1 for scalar fields)
- `P`: Number of Gauss-Lobatto nodes in one dimension. The polynomial degree of the resulting Q_k element is k=P-1.

- `Q`: Number of quadrature points in one dimension.
- `qmode`: Distribution of the Q quadrature points (affects order of accuracy for the quadrature)
- `[out] basis`: Address of the variable where the newly created CeedBasis will be stored.

int **`CeedBasisCreateH1`**(*Ceed ceed*, *CeedElemTopology topo*, *CeedInt ncomp*, *CeedInt nnodes*, *CeedInt nqpts*, **`const`** *CeedScalar *interp*, **`const`** *CeedScalar *grad*, **`const`** *CeedScalar *qref*, **`const`** *CeedScalar *qweight*, *CeedBasis *basis*)

Create a non tensor-product basis for H^1 discretizations.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedBasis will be created
- `topo`: Topology of element, e.g. hypercube, simplex, ect
- `ncomp`: Number of field components (1 for scalar fields)
- `nnodes`: Total number of nodes
- `nqpts`: Total number of quadrature points
- `interp`: Row-major (nqpts * nnodes) matrix expressing the values of nodal basis functions at quadrature points
- `grad`: Row-major (nqpts * dim * nnodes) matrix expressing derivatives of nodal basis functions at quadrature points
- `qref`: Array of length nqpts holding the locations of quadrature points on the reference element [-1, 1]
- `qweight`: Array of length nqpts holding the quadrature weights on the reference element
- `[out] basis`: Address of the variable where the newly created CeedBasis will be stored.

int **`CeedBasisView`**(*CeedBasis basis*, FILE **stream*)

View a CeedBasis.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis to view
- `stream`: Stream to view to, e.g., stdout

int **`CeedBasisApply`**(*CeedBasis basis*, *CeedInt nelem*, *CeedTransposeMode tmode*, *CeedEvalMode emode*, *CeedVector u*, *CeedVector v*)

Apply basis evaluation from nodes to quadrature points or vice versa.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis to evaluate
- `nelem`: The number of elements to apply the basis evaluation to; the backend will specify the ordering in *CeedElemRestrictionCreateBlocked()*

- `tmode`: *CEED_NOTRANSPOSE* to evaluate from nodes to quadrature points, *CEED_TRANSPOSE* to apply the transpose, mapping from quadrature points to nodes
- `emode`: *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_WEIGHT* to use quadrature weights.
- `[in] u`: Input CeedVector
- `[out] v`: Output CeedVector

int **`CeedBasisGetDimension`**(*CeedBasis basis*, *CeedInt *dim*)

Get dimension for given CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] dim`: Variable to store dimension of basis

int **`CeedBasisGetTopology`**(*CeedBasis basis*, *CeedElemTopology *topo*)

Get topology for given CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] topo`: Variable to store topology of basis

int **`CeedBasisGetNumComponents`**(*CeedBasis basis*, *CeedInt *numcomp*)

Get number of components for given CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] numcomp`: Variable to store number of components of basis

int **`CeedBasisGetNumNodes`**(*CeedBasis basis*, *CeedInt *P*)

Get total number of nodes (in dim dimensions) of a CeedBasis.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] P`: Variable to store number of nodes

int **`CeedBasisGetNumNodes1D`**(*CeedBasis basis*, *CeedInt *P1d*)

Get total number of nodes (in 1 dimension) of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] P1d`: Variable to store number of nodes

int **`CeedBasisGetNumQuadraturePoints`**(*CeedBasis basis*, *CeedInt *Q*)

Get total number of quadrature points (in dim dimensions) of a CeedBasis.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] Q`: Variable to store number of quadrature points

int **`CeedBasisGetNumQuadraturePoints1D`**(*CeedBasis basis*, *CeedInt *Q1d*)

Get total number of quadrature points (in 1 dimension) of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] Q1d`: Variable to store number of quadrature points

int **`CeedBasisGetQRef`**(*CeedBasis basis*, **`const`** *CeedScalar **qref*)

Get reference coordinates of quadrature points (in dim dimensions) of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] qref`: Variable to store reference coordinates of quadrature points

int **`CeedBasisGetQWeights`**(*CeedBasis basis*, **`const`** *CeedScalar **qweight*)

Get quadrature weights of quadrature points (in dim dimensions) of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] qweight`: Variable to store quadrature weights

int **`CeedBasisGetInterp`**(*CeedBasis basis*, **`const`** *CeedScalar **interp*)

Get interpolation matrix of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] interp`: Variable to store interpolation matrix

int **`CeedBasisGetInterp1D`**(*CeedBasis basis*, **`const`** *CeedScalar **interp1d*)

Get 1D interpolation matrix of a tensor product CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] interp1d`: Variable to store interpolation matrix

int **`CeedBasisGetGrad`**(*CeedBasis basis*, **`const`** *CeedScalar **grad*)

Get gradient matrix of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] grad`: Variable to store gradient matrix

int **`CeedBasisGetGrad1D`**(*CeedBasis basis*, **`const`** *CeedScalar **grad1d*)

Get 1D gradient matrix of a tensor product CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] grad1d`: Variable to store gradient matrix

int **`CeedBasisDestroy`**(*CeedBasis *basis*)

Destroy a CeedBasis.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis to destroy

int **`CeedGaussQuadrature`**(*CeedInt Q*, *CeedScalar *qref1d*, *CeedScalar *qweight1d*)

Construct a Gauss-Legendre quadrature.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `Q`: Number of quadrature points (integrates polynomials of degree 2*Q-1 exactly)
- `[out] qref1d`: Array of length Q to hold the abscissa on [-1, 1]
- `[out] qweight1d`: Array of length Q to hold the weights

int **`CeedLobattoQuadrature`**(*CeedInt Q*, *CeedScalar *qref1d*, *CeedScalar *qweight1d*)

Construct a Gauss-Legendre-Lobatto quadrature.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `Q`: Number of quadrature points (integrates polynomials of degree 2*Q-3 exactly)
- `[out] qref1d`: Array of length Q to hold the abscissa on [-1, 1]
- `[out] qweight1d`: Array of length Q to hold the weights

int **`CeedQRFactorization`**(*Ceed ceed*, *CeedScalar *mat*, *CeedScalar *tau*, *CeedInt m*, *CeedInt n*)

Return QR Factorization of a matrix.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed context for error handling
- `[inout] mat`: Row-major matrix to be factorized in place
- `[inout] tau`: Vector of length m of scaling factors
- `m`: Number of rows
- `n`: Number of columns

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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed context for error handling
- `[inout] mat`: Row-major matrix to be factorized in place
- `[out] lambda`: Vector of length n of eigenvalues
- `n`: Number of rows/columns

int **`CeedSimultaneousDiagonalization`**(*Ceed ceed*, *CeedScalar *matA*, *CeedScalar *matB*, *Ceed-Scalar *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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed context for error handling
- `[in] matA`: Row-major matrix to be factorized with eigenvalues
- `[in] matB`: Row-major matrix to be factorized to identity
- `[out] x`: Row-major orthogonal matrix
- `[out] lambda`: Vector of length n of generalized eigenvalues

- `n`: Number of rows/columns

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.

Modes can be bitwise ORed when passing to most functions.

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 a nodal basis.

`enumerator CEED_EVAL_DIV`

Evaluate divergence at quadrature points from input in a nodal basis.

`enumerator CEED_EVAL_CURL`

Evaluate curl at quadrature points from input in a nodal 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 H1 element basis.

Dimension can be extracted with bitwise AND (CeedElemTopology & 2**(dim + 2)) == TRUE

Values:

`enumerator CEED_LINE`

Line.

`enumerator CEED_TRIANGLE`

Triangle - 2D shape.

enumerator CEED_QUAD
Quadralateral - 2D shape.

enumerator CEED_TET
Tetrahedron - 3D shape.

enumerator CEED_PYRAMID
Pyramid - 3D shape.

enumerator CEED_PRISM
Prism - 3D shape.

enumerator CEED_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.

const *CeedQFunction* **CEED_QFUNCTION_NONE** = &ceed_qfunction_none

int **CeedQFunctionCreateInterior**(*Ceed ceed*, *CeedInt vlength*, 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.

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedQFunction will be created
- `vlength`: Vector length. Caller must ensure that number of quadrature points is a multiple of vlength.
- `f`: Function pointer to evaluate action at quadrature points. See Public API for CeedQFunction.
- `source`: Absolute path to source of QFunction, "\abs_path\file.h:function_name". For support across all backends, this source must only contain constructs supported by C99, C++11, and CUDA.
- `[out] qf`: Address of the variable where the newly created CeedQFunction will be stored

User Functions

int **CeedQFunctionCreateInteriorByName**(*Ceed ceed*, **const** char **name*, *CeedQFunction *qf*)
Create a CeedQFunction for evaluating interior (volumetric) terms by name.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedQFunction will be created
- `name`: Name of QFunction to use from gallery
- `[out] qf`: Address of the variable where the newly created CeedQFunction will be stored

int **`CeedQFunctionCreateIdentity`**(*Ceed ceed*, *CeedInt size*, *CeedEvalMode inmode*, *CeedEvalMode outmode*, *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 CeedRestriction 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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedQFunction will be created
- `[in] size`: Size of the qfunction fields
- `[in] inmode`: CeedEvalMode for input to CeedQFunction
- `[in] outmode`: CeedEvalMode for output to CeedQFunction
- `[out] qf`: Address of the variable where the newly created CeedQFunction will be stored

int **`CeedQFunctionAddInput`**(*CeedQFunction qf*, **`const`** char **fieldname*, *CeedInt size*, *CeedEvalMode emode*)

Add a CeedQFunction input.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `fieldname`: Name of QFunction field
- `size`: Size of QFunction field, (ncomp * dim) for *CEED_EVAL_GRAD* or (ncomp * 1) for *CEED_EVAL_NONE* and *CEED_EVAL_INTERP*
- `emode`: *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients.

int **`CeedQFunctionAddOutput`**(*CeedQFunction qf*, **`const`** char **fieldname*, *CeedInt size*, *CeedEvalMode emode*)

Add a CeedQFunction output.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `fieldname`: Name of QFunction field

- `size`: Size of QFunction field, (ncomp * dim) for *CEED_EVAL_GRAD* or (ncomp * 1) for *CEED_EVAL_NONE* and *CEED_EVAL_INTERP*
- `emode`: *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients.

int **CeedQFunctionSetContext**(*CeedQFunction qf*, *CeedQFunctionContext ctx*)

Set global context for a CeedQFunction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `ctx`: Context data to set

int **CeedQFunctionView**(*CeedQFunction qf*, FILE **stream*)

View a CeedQFunction.

User Functions

Return Error code: 0 - success, otherwise - failure

Parameters

- `[in] qf`: CeedQFunction to view
- `[in] stream`: Stream to write; typically stdout/stderr or a file

int **CeedQFunctionApply**(*CeedQFunction qf*, *CeedInt Q*, *CeedVector *u*, *CeedVector *v*)

Apply the action of a CeedQFunction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `Q`: Number of quadrature points
- `[in] u`: Array of input CeedVectors
- `[out] v`: Array of output CeedVectors

int **CeedQFunctionDestroy**(*CeedQFunction *qf*)

Destroy a CeedQFunction.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction to destroy

int **CeedQFunctionContextCreate**(*Ceed ceed*, *CeedQFunctionContext *ctx*)

Create a CeedQFunctionContext for storing CeedQFunction user context data.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedQFunctionContext will be created
- `[out] ctx`: Address of the variable where the newly created CeedQFunctionContext will be stored

int **`CeedQFunctionContextSetData`**(*CeedQFunctionContext ctx*, *CeedMemType mtype*, *CeedCopyMode cmode*, 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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext
- `mtype`: Memory type of the data being passed
- `cmode`: Copy mode for the data
- `data`: Data to be used

int **`CeedQFunctionContextGetData`**(*CeedQFunctionContext ctx*, *CeedMemType mtype*, 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.

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext to access
- `mtype`: Memory type on which to access the data. If the backend uses a different memory type, this will perform a copy.
- `[out] data`: Data on memory type mtype

int **`CeedQFunctionContextRestoreData`**(*CeedQFunctionContext ctx*, void **data*)

Restore data obtained using *CeedQFunctionContextGetData()*

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext to restore
- `data`: Data to restore

int **`CeedQFunctionContextView`**(*CeedQFunctionContext ctx*, FILE **stream*)

View a CeedQFunctionContext.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] ctx`: CeedQFunctionContext to view
- `[in] stream`: Filestream to write to

int **`CeedQFunctionContextDestroy`**(*CeedQFunctionContext *ctx*)

Destroy a CeedQFunctionContext.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext to destroy

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.

`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 $$ E^T B^T f(B E u) $$ 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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedOperator will be created
- `qf`: QFunction defining the action of the operator at quadrature points
- `dqf`: QFunction defining the action of the Jacobian of *qf* (or *CEED_QFUNCTION_NONE*)

- `dqfT`: QFunction defining the action of the transpose of the Jacobian of *qf* (or *CEED_QFUNCTION_NONE*)
- `[out] op`: Address of the variable where the newly created CeedOperator will be stored

int **`CeedCompositeOperatorCreate`**(*Ceed ceed*, *CeedOperator *op*)

Create an operator that composes the action of several operators.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedOperator will be created
- `[out] op`: Address of the variable where the newly created Composite CeedOperator will be stored

int **`CeedOperatorSetField`**(*CeedOperator op*, **`const`** char **fieldname*, *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 and at most one active output.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator on which to provide the field
- `fieldname`: Name of the field (to be matched with the name used by CeedQFunction)
- `r`: CeedElemRestriction
- `b`: CeedBasis in which the field resides or *CEED_BASIS_COLLOCATED* if collocated with quadrature points
- `v`: 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

int **`CeedCompositeOperatorAddSub`**(*CeedOperator compositeop*, *CeedOperator subop*)

Add a sub-operator to a composite CeedOperator.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] compositeop`: Composite CeedOperator
- `subop`: Sub-operator CeedOperator

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].

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to assemble CeedQFunction
- `[out] assembled`: CeedVector to store assembled CeedQFunction at quadrature points
- `[out] rstr`: CeedElemRestriction for CeedVector containing assembled CeedQFunction
- `request`: Address of CeedRequest for non-blocking completion, else *CEED_REQUEST_IMMEDIATE*

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.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to assemble CeedQFunction
- `[out] assembled`: CeedVector to store assembled CeedOperator diagonal
- `request`: Address of CeedRequest for non-blocking completion, else *CEED_REQUEST_IMMEDIATE*

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.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to assemble CeedQFunction
- `[out] assembled`: CeedVector to store assembled CeedOperator diagonal
- `request`: Address of CeedRequest for non-blocking completion, else *CEED_REQUEST_IMMEDIATE*

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.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to assemble CeedQFunction
- `[out] assembled`: CeedVector to store assembled CeedOperator point block diagonal, provided in row-major form with an *ncomp* * *ncomp* 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`: Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

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.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to assemble CeedQFunction
- `[out] assembled`: CeedVector to store assembled CeedOperator point block diagonal, provided in row-major form with an *ncomp* * *ncomp* 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`: Address of CeedRequest for non-blocking completion, else CEED_REQUEST_IMMEDIATE

int **CeedOperatorMultigridLevelCreate**(*CeedOperator opFine*, *CeedVector PMultFine*, *CeedElemRestriction rstrCoarse*, *CeedBasis basisCoarse*, *CeedOperator *opCoarse*, *CeedOperator *opProlong*, *CeedOperator *opRestrict*)

Create a multigrid coarse operator and level transfer operators for a CeedOperator, creating the prolongation basis from the fine and coarse grid interpolation.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] opFine`: Fine grid operator
- `[in] PMultFine`: L-vector multiplicity in parallel gather/scatter

- `[in] rstrCoarse`: Coarse grid restriction
- `[in] basisCoarse`: Coarse grid active vector basis
- `[out] opCoarse`: Coarse grid operator
- `[out] opProlong`: Coarse to fine operator
- `[out] opRestrict`: Fine to coarse operator

int **`CeedOperatorMultigridLevelCreateTensorH1`**(*CeedOperator opFine*, *CeedVector PMultFine*, *CeedElemRestriction rstrCoarse*, *CeedBasis basisCoarse*, **`const`** *CeedScalar *interpCtoF*, *CeedOperator *opCoarse*, *CeedOperator *opProlong*, *CeedOperator *opRestrict*)

Create a multigrid coarse operator and level transfer operators for a CeedOperator with a tensor basis for the active basis.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] opFine`: Fine grid operator
- `[in] PMultFine`: L-vector multiplicity in parallel gather/scatter
- `[in] rstrCoarse`: Coarse grid restriction
- `[in] basisCoarse`: Coarse grid active vector basis
- `[in] interpCtoF`: Matrix for coarse to fine interpolation
- `[out] opCoarse`: Coarse grid operator
- `[out] opProlong`: Coarse to fine operator
- `[out] opRestrict`: Fine to coarse operator

int **`CeedOperatorMultigridLevelCreateH1`**(*CeedOperator opFine*, *CeedVector PMultFine*, *CeedElemRestriction rstrCoarse*, *CeedBasis basisCoarse*, **`const`** *CeedScalar *interpCtoF*, *CeedOperator *opCoarse*, *CeedOperator *opProlong*, *CeedOperator *opRestrict*)

Create a multigrid coarse operator and level transfer operators for a CeedOperator with a non-tensor basis for the active vector.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] opFine`: Fine grid operator
- `[in] PMultFine`: L-vector multiplicity in parallel gather/scatter
- `[in] rstrCoarse`: Coarse grid restriction
- `[in] basisCoarse`: Coarse grid active vector basis
- `[in] interpCtoF`: Matrix for coarse to fine interpolation
- `[out] opCoarse`: Coarse grid operator
- `[out] opProlong`: Coarse to fine operator

- `[out] opRestrict`: Fine to coarse operator

int **`CeedOperatorCreateFDMElementInverse`**(*CeedOperator op*, *CeedOperator *fdminv*, *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 S^hat V. The CeedOperator must be linear and non-composite. The associated CeedQFunction must therefore also be linear.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to create element inverses
- `[out] fdminv`: CeedOperator to apply the action of a FDM based inverse for each element
- `request`: Address of CeedRequest for non-blocking completion, else *CEED_REQUEST_IMMEDIATE*

int **`CeedOperatorView`**(*CeedOperator op*, FILE **stream*)

View a CeedOperator.

User Functions

Return Error code: 0 - success, otherwise - failure

Parameters

- `[in] op`: CeedOperator to view
- `[in] stream`: Stream to write; typically stdout/stderr or a file

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()*.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: 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`: Address of CeedRequest for non-blocking completion, else *CEED_REQUEST_IMMEDIATE*

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

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: 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`: Address of CeedRequest for non-blocking completion, else *CEED_REQUEST_IMMEDIATE*

int **`CeedOperatorDestroy`**(*CeedOperator *op*)

Destroy a CeedOperator.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to destroy

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

void **`CeedDebugImpl`**(**`const`** *Ceed ceed*, **`const`** char **format*, ...)

Print Ceed debugging information.

Backend Developer Functions

Return None

Parameters

- `ceed`: Ceed context
- `format`: Printing format

void **`CeedDebugImpl256`**(**`const`** *Ceed ceed*, **`const`** unsigned char *color*, **`const`** char **format*, ...)

Print Ceed debugging information in color.

Backend Developer Functions

Return None

Parameters

- `ceed`: Ceed context
- `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

Return An error code: 0 - success, otherwise - failure

See *CeedFree()*

Parameters

- `n`: Number of units to allocate
- `unit`: Size of each unit
- `p`: Address of pointer to hold the result.

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

Return An error code: 0 - success, otherwise - failure

See *CeedFree()*

Parameters

- `n`: Number of units to allocate
- `unit`: Size of each unit
- `p`: Address of pointer to hold the result.

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

Return An error code: 0 - success, otherwise - failure

See *CeedFree()*

Parameters

- `n`: Number of units to allocate
- `unit`: Size of each unit
- `p`: Address of pointer to hold the result.

int **`CeedFree`**(void **p*)

Free memory allocated using CeedMalloc() or CeedCalloc()

Parameters

- `p`: 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

Return An error code: 0 - success, otherwise - failure

Parameters

- `prefix`: Prefix of resources for this backend to respond to. For example, the reference backend responds to "/cpu/self".
- `init`: Initialization function called by *CeedInit()* when the backend is selected to drive the requested resource.
- `priority`: Integer priority. Lower values are preferred in case the resource requested by *CeedInit()* has non-unique best prefix match.

int **CeedIsDebug**(*Ceed ceed*, bool **isDebug*)
Return debugging status flag.

Bcakend

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to get debugging flag
- `isDebug`: Variable to store debugging flag

int **CeedGetParent**(*Ceed ceed*, *Ceed *parent*)
Retrieve a parent Ceed context.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to retrieve parent of
- `[out] parent`: Address to save the parent to

int **CeedGetDelegate**(*Ceed ceed*, *Ceed *delegate*)
Retrieve a delegate Ceed context.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to retrieve delegate of
- `[out] delegate`: Address to save the delegate to

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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to set delegate of
- `[out] delegate`: Address to set the delegate to

int **`CeedGetObjectDelegate`**(*Ceed ceed*, *Ceed *delegate*, **`const`** char **objname*)

Retrieve a delegate Ceed context for a specific object type.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to retrieve delegate of
- `[out] delegate`: Address to save the delegate to
- `[in] objname`: Name of the object type to retrieve delegate for

int **`CeedSetObjectDelegate`**(*Ceed ceed*, *Ceed delegate*, **`const`** char **objname*)

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, refceed, "Basis") uses refceed implementations for all CeedBasis backend functions.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to set delegate of
- `[out] delegate`: Address to set the delegate to
- `[in] objname`: Name of the object type to set delegate for

int **`CeedGetOperatorFallbackResource`**(*Ceed ceed*, **`const`** char ***resource*)

Get the fallback resource for CeedOperators.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context
- `[out] resource`: Variable to store fallback resource

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

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] ceed`: Ceed context
- `resource`: Fallback resource to set

int **CeedGetOperatorFallbackParentCeed**(*Ceed ceed*, *Ceed *parent*)

Get the parent Ceed context associated with a fallback Ceed context for a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context
- `[out] parent`: Variable to store parent Ceed context

int **CeedSetDeterministic**(*Ceed ceed*, bool *isDeterministic*)

Flag Ceed context as deterministic.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed to flag as deterministic

int **CeedSetBackendFunction**(*Ceed ceed*, **const** char **type*, void **object*, **const** char **fname*, 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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context for error handling
- `type`: Type of Ceed object to set function for
- `[out] object`: Ceed object to set function for
- `fname`: Name of function to set
- `f`: Function to set

int **CeedGetData**(*Ceed ceed*, void **data*)

Retrieve backend data for a Ceed context.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to retrieve data of
- `[out] data`: Address to save data to

int **CeedSetData**(*Ceed ceed*, void **data*)

Set backend data for a Ceed context.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: Ceed context to set data of
- `data`: Address of data to set

6.2.2 CeedVector

int **`CeedVectorGetCeed`**(*CeedVector vec*, *Ceed *ceed*)

Get the Ceed associated with a CeedVector.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to retrieve state
- `[out] ceed`: Variable to store ceed

int **`CeedVectorGetState`**(*CeedVector vec*, uint64_t **state*)

Get the state of a CeedVector.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to retrieve state
- `[out] state`: Variable to store state

int **`CeedVectorAddReference`**(*CeedVector vec*)

Add a reference to a CeedVector.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] vec`: CeedVector to increment reference counter

int **`CeedVectorGetData`**(*CeedVector vec*, void **data*)

Get the backend data of a CeedVector.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `vec`: CeedVector to retrieve state
- `[out] data`: Variable to store data

int **`CeedVectorSetData`**(*CeedVector vec*, void **data*)

Set the backend data of a CeedVector.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] vec`: CeedVector to retrieve state
- `data`: Data to set

6.2.3 CeedElemRestriction

int **`CeedElemRestrictionGetCeed`**(*CeedElemRestriction rstr*, *Ceed *ceed*)

Get the Ceed associated with a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] ceed`: Variable to store Ceed

int **`CeedElemRestrictionGetStrides`**(*CeedElemRestriction rstr*, *CeedInt* (**strides*)[3])

Get the strides of a strided CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] strides`: Variable to store strides array

int **`CeedElemRestrictionGetOffsets`**(*CeedElemRestriction rstr*, *CeedMemType mtype*, **`const`** *CeedInt **offsets*)

Get read-only access to a CeedElemRestriction offsets array by memtype.

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction to retrieve offsets
- `mtype`: Memory type on which to access the array. If the backend uses a different memory type, this will perform a copy (possibly cached).
- `[out] offsets`: Array on memory type mtype

int **`CeedElemRestrictionRestoreOffsets`**(*CeedElemRestriction rstr*, **`const`** *CeedInt **offsets*)

Restore an offsets array obtained using *CeedElemRestrictionGetOffsets()*

User Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction to restore
- `offsets`: Array of offset data

int **`CeedElemRestrictionIsStrided`**(*CeedElemRestriction rstr*, bool **isstrided*)

Get the strided status of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] isstrided`: Variable to store strided status, 1 if strided else 0

int **`CeedElemRestrictionHasBackendStrides`**(*CeedElemRestriction rstr*, bool **hasbackendstrides*)

Get the backend stride status of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] status`: Variable to store stride status

int **`CeedElemRestrictionGetELayout`**(*CeedElemRestriction rstr*, *CeedInt* (**layout*)[3])

Get the E-vector layout of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] layout`: 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]

int **`CeedElemRestrictionSetELayout`**(*CeedElemRestriction rstr*, *CeedInt layout*[3])

Set the E-vector layout of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `layout`: 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]

int **`CeedElemRestrictionGetData`**(*CeedElemRestriction rstr*, void **data*)

Get the backend data of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `rstr`: CeedElemRestriction
- `[out] data`: Variable to store data

int **`CeedElemRestrictionSetData`**(*CeedElemRestriction rstr*, void **data*)

Set the backend data of a CeedElemRestriction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] rstr`: CeedElemRestriction
- `data`: Data to set

6.2.4 CeedBasis

int **`CeedBasisGetCollocatedGrad`**(*CeedBasis basis*, *CeedScalar *collograd1d*)

Return collocated grad matrix.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] collograd1d`: Row-major (Q1d * Q1d) matrix expressing derivatives of basis functions at quadrature points

int **`CeedBasisGetCeed`**(*CeedBasis basis*, *Ceed *ceed*)

Get Ceed associated with a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] ceed`: Variable to store Ceed

int **`CeedBasisIsTensor`**(*CeedBasis basis*, bool **istensor*)

Get tensor status for given CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] istensor`: Variable to store tensor status

int **`CeedBasisGetData`**(*CeedBasis basis*, void **data*)

Get backend data of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] data`: Variable to store data

int **`CeedBasisSetData`**(*CeedBasis basis*, void **data*)

Set backend data of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] basis`: CeedBasis
- `data`: Data to set

int **`CeedBasisGetTopologyDimension`**(*CeedElemTopology topo*, *CeedInt *dim*)

Get dimension for given CeedElemTopology.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `topo`: CeedElemTopology
- `[out] dim`: Variable to store dimension of topology

int **`CeedBasisGetTensorContract`**(*CeedBasis basis*, CeedTensorContract **contract*)

Get CeedTensorContract of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `basis`: CeedBasis
- `[out] contract`: Variable to store CeedTensorContract

int **`CeedBasisSetTensorContract`**(*CeedBasis basis*, CeedTensorContract **contract*)

Set CeedTensorContract of a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] basis`: CeedBasis
- `contract`: CeedTensorContract to set

int **`CeedMatrixMultiply`**(*Ceed ceed*, **`const`** *CeedScalar *matA*, **`const`** *CeedScalar *matB*, *CeedScalar *matC*, *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

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed context for error handling
- `[in] matA`: Row-major matrix A
- `[in] matB`: Row-major matrix B
- `[out] matC`: Row-major output matrix C
- `m`: Number of rows of C

- `n`: Number of columns of C
- `kk`: Number of columns of A/rows of B

int **`CeedTensorContractCreate`**(*Ceed* *ceed*, *CeedBasis* *basis*, CeedTensorContract **contract*)

Create a CeedTensorContract object for a CeedBasis.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ceed`: A Ceed object where the CeedTensorContract will be created
- `basis`: CeedBasis for which the tensor contraction will be used
- `[out] contract`: Address of the variable where the newly created CeedTensorContract will be stored.

int **`CeedTensorContractApply`**(CeedTensorContract *contract*, *CeedInt* *A*, *CeedInt* *B*, *CeedInt* *C*, *CeedInt* *J*, **`const`** *CeedScalar* ***`restrict`** *t*, *CeedTransposeMode* *tmode*, **`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

Return An error code: 0 - success, otherwise - failure

Parameters

- `contract`: CeedTensorContract to use
- `A`: First index of u, v
- `B`: Middle index of u, one index of t
- `C`: Last index of u, v
- `J`: Middle index of v, one index of t
- `[in] t`: Tensor array to contract against
- `tmode`: Transpose mode for t, *CEED_NOTRANSPOSE* for t_jb *CEED_TRANSPOSE* for t_bj
- `add`: Add mode
- `[in] u`: Input array
- `[out] v`: Output array

int **`CeedTensorContractGetCeed`**(CeedTensorContract *contract*, *Ceed* **ceed*)

Get Ceed associated with a CeedTensorContract.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `contract`: CeedTensorContract
- `[out] ceed`: Variable to store Ceed

int **CeedTensorContractGetData**(CeedTensorContract *contract*, void **data*)
Get backend data of a CeedTensorContract.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `contract`: CeedTensorContract
- `[out] data`: Variable to store data

int **CeedTensorContractSetData**(CeedTensorContract *contract*, void **data*)
Set backend data of a CeedTensorContract.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] contract`: CeedTensorContract
- `data`: Data to set

int **CeedTensorContractDestroy**(CeedTensorContract **contract*)
Destroy a CeedTensorContract.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `contract`: CeedTensorContract to destroy

6.2.5 CeedQFunction

typedef struct CeedQFunctionField_private ***CeedQFunctionField**
Handle for object describing CeedQFunction fields.

int **CeedQFunctionGetCeed**(*CeedQFunction qf*, *Ceed *ceed*)
Get the Ceed associated with a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] ceed`: Variable to store Ceed

int **CeedQFunctionGetVectorLength**(*CeedQFunction qf*, *CeedInt *vlength*)
Get the vector length of a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] vlength`: Variable to store vector length

int **CeedQFunctionGetNumArgs**(*CeedQFunction qf*, *CeedInt *numinput*, *CeedInt *numoutput*)

Get the number of inputs and outputs to a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] numinput`: Variable to store number of input fields
- `[out] numoutput`: Variable to store number of output fields

int **CeedQFunctionGetSourcePath**(*CeedQFunction qf*, char ***source*)

Get the source path string for a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] source`: Variable to store source path string

int **CeedQFunctionGetUserFunction**(*CeedQFunction qf*, CeedQFunctionUser **f*)

Get the User Function for a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] f`: Variable to store user function

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 CeedQ-FunctionContext.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] ctx`: Variable to store CeedQFunctionContext

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.

Return An error code: 0 - success, otherwise - failure Backend Developer Functions

Parameters

- `qf`: CeedQFunction
- `[out] ctx`: Variable to store CeedQFunctionContext

int **`CeedQFunctionIsIdentity`**(*CeedQFunction qf*, bool **isidentity*)

Determine if QFunction is identity.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] isidentity`: Variable to store identity status

int **`CeedQFunctionGetData`**(*CeedQFunction qf*, void **data*)

Get backend data of a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] data`: Variable to store data

int **`CeedQFunctionSetData`**(*CeedQFunction qf*, void **data*)

Set backend data of a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] qf`: CeedQFunction
- `data`: Data to set

int **`CeedQFunctionGetFields`**(*CeedQFunction qf*, *CeedQFunctionField **inputfields*, *CeedQFunction-Field **outputfields*)

Get the CeedQFunctionFields of a CeedQFunction.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `[out] inputfields`: Variable to store inputfields
- `[out] outputfields`: Variable to store outputfields

int **`CeedQFunctionFieldGetName`**(*CeedQFunctionField qffield*, char ***fieldname*)

Get the name of a CeedQFunctionField.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qffield`: CeedQFunctionField
- `[out] fieldname`: Variable to store the field name

int **`CeedQFunctionFieldGetSize`**(*CeedQFunctionField qffield*, *CeedInt *size*)

Get the number of components of a CeedQFunctionField.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qffield`: CeedQFunctionField
- `[out] size`: Variable to store the size of the field

int **`CeedQFunctionFieldGetEvalMode`**(*CeedQFunctionField qffield*, *CeedEvalMode *emode*)

Get the CeedEvalMode of a CeedQFunctionField.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qffield`: CeedQFunctionField
- `[out] emode`: Variable to store the field evaluation mode

int **`CeedQFunctionContextGetCeed`**(*CeedQFunctionContext ctx*, *Ceed *ceed*)

Get the Ceed associated with a CeedQFunctionContext.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext
- `[out] ceed`: Variable to store Ceed

int **`CeedQFunctionContextGetState`**(*CeedQFunctionContext ctx*, uint64_t **state*)

Get the state of a CeedQFunctionContext.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext to retrieve state
- `[out] state`: Variable to store state

int **`CeedQFunctionContextGetContextSize`**(*CeedQFunctionContext ctx*, size_t **ctxsize*)

Get data size for a Context.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext
- `[out] ctxsize`: Variable to store size of context data values

int **`CeedQFunctionContextGetBackendData`**(*CeedQFunctionContext ctx*, void **data*)

Get backend data of a CeedQFunctionContext.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `ctx`: CeedQFunctionContext
- `[out] data`: Variable to store data

int **`CeedQFunctionContextSetBackendData`**(*CeedQFunctionContext ctx*, void **data*)

Set backend data of a CeedQFunctionContext.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] ctx`: CeedQFunctionContext
- `data`: Data to set

6.2.6 CeedOperator

`typedef struct` CeedOperatorField_private ***`CeedOperatorField`**

Handle for object describing CeedOperator fields.

int **`CeedOperatorGetCeed`**(*CeedOperator op*, *Ceed *ceed*)

Get the Ceed associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] ceed`: Variable to store Ceed

int **`CeedOperatorGetNumElements`**(*CeedOperator op*, *CeedInt *numelem*)

Get the number of elements associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] numelem`: Variable to store number of elements

int **`CeedOperatorGetNumQuadraturePoints`**(*CeedOperator op*, *CeedInt *numqpts*)

Get the number of quadrature points associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] numqpts`: Variable to store vector number of quadrature points

int **CeedOperatorGetNumArgs**(*CeedOperator op*, *CeedInt *numargs*)

Get the number of arguments associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] numargs`: Variable to store vector number of arguments

int **CeedOperatorIsSetupDone**(*CeedOperator op*, bool **issetupdone*)

Get the setup status of a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] issetupdone`: Variable to store setup status

int **CeedOperatorGetQFunction**(*CeedOperator op*, *CeedQFunction *qf*)

Get the QFunction associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] qf`: Variable to store QFunction

int **CeedOperatorIsComposite**(*CeedOperator op*, bool **iscomposite*)

Get a boolean value indicating if the CeedOperator is composite.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] iscomposite`: Variable to store composite status

int **CeedOperatorGetNumSub**(*CeedOperator op*, *CeedInt *numsub*)

Get the number of suboperators associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] numsub`: Variable to store number of suboperators

int **CeedOperatorGetSubList**(*CeedOperator op*, *CeedOperator **suboperators*)

Get the list of suboperators associated with a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] suboperators`: Variable to store list of suboperators

int **`CeedOperatorGetData`**(*CeedOperator op*, void **data*)

Get the backend data of a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] data`: Variable to store data

int **`CeedOperatorSetData`**(*CeedOperator op*, void **data*)

Set the backend data of a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[out] op`: CeedOperator
- `data`: Data to set

int **`CeedOperatorSetSetupDone`**(*CeedOperator op*)

Set the setup flag of a CeedOperator to True.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator

int **`CeedOperatorGetFields`**(*CeedOperator op*, *CeedOperatorField **inputfields*, *CeedOperatorField **outputfields*)

Get the CeedOperatorFields of a CeedOperator.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator
- `[out] inputfields`: Variable to store inputfields
- `[out] outputfields`: Variable to store outputfields

int **`CeedOperatorFieldGetElemRestriction`**(*CeedOperatorField opfield*, *CeedElemRestriction *rstr*)

Get the CeedElemRestriction of a CeedOperatorField.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `opfield`: CeedOperatorField
- `[out] rstr`: Variable to store CeedElemRestriction

int **`CeedOperatorFieldGetBasis`**(*CeedOperatorField opfield*, *CeedBasis *basis*)

Get the CeedBasis of a CeedOperatorField.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `opfield`: CeedOperatorField
- `[out] basis`: Variable to store CeedBasis

int **`CeedOperatorFieldGetVector`**(*CeedOperatorField opfield*, *CeedVector *vec*)

Get the CeedVector of a CeedOperatorField.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `opfield`: CeedOperatorField
- `[out] vec`: Variable to store CeedVector

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

6.3.2 CeedVector

6.3.3 CeedElemRestriction

int **`CeedPermutePadOffsets`**(**const** *CeedInt *offsets*, *CeedInt *blkoffsets*, *CeedInt nblk*, *CeedInt nelem*, *CeedInt blksize*, *CeedInt elemsize*)

Permute and pad offsets for a blocked restriction.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `offsets`: Array of shape [*nelem*, *elemsize*]. Row i holds the ordered list of the offsets (into the input CeedVector) for the unknowns corresponding to element i, where 0 <= i < *nelem*. All offsets must be in the range [0, *lsize* - 1].
- `blkoffsets`: Array of permuted and padded offsets of shape [*nblk*, *elemsize*, *blksize*].
- `nblk`: Number of blocks
- `nelem`: Number of elements
- `blksize`: Number of elements in a block

- `elemsize`: Size of each element

6.3.4 CeedBasis

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

Return An error code: 0 - success, otherwise - failure

Parameters

- `[inout] A`: Matrix to apply Householder reflection to, in place
- `v`: Householder vector
- `b`: Scaling factor
- `m`: Number of rows in A
- `n`: Number of columns in A
- `row`: Row stride
- `col`: Col stride

int **CeedHouseholderApplyQ**(*CeedScalar *A*, **const** *CeedScalar *Q*, **const** *CeedScalar *tau*, *CeedTransposeMode tmode*, *CeedInt m*, *CeedInt n*, *CeedInt k*, *CeedInt row*, *CeedInt col*)

Apply Householder Q matrix.

Compute A = Q A where Q is mxm and A is mxn.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[inout] A`: Matrix to apply Householder Q to, in place
- `Q`: Householder Q matrix
- `tau`: Householder scaling factors
- `tmode`: Transpose mode for application
- `m`: Number of rows in A
- `n`: Number of columns in A
- `k`: Number of elementary reflectors in Q, k<m
- `row`: Row stride in A
- `col`: Col stride in A

int **CeedGivensRotation**(*CeedScalar *A*, *CeedScalar c*, *CeedScalar s*, *CeedTransposeMode tmode*, *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

Return An error code: 0 - success, otherwise - failure

Parameters

- `[inout] A`: Row major matrix to apply Givens rotation to, in place
- `c`: Cosine factor
- `s`: Sine factor
- `tmode`: *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`: First row/column to apply rotation
- `k`: Second row/column to apply rotation
- `m`: Number of rows in A
- `n`: Number of columns in A

int **`CeedScalarView`**(**`const`** char **name*, **`const`** char **fpformat*, *CeedInt m*, *CeedInt n*, **`const`** *CeedScalar *a*, FILE **stream*)

View an array stored in a CeedBasis.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] name`: Name of array
- `[in] fpformat`: Printing format
- `[in] m`: Number of rows in array
- `[in] n`: Number of columns in array
- `[in] a`: Array to be viewed
- `[in] stream`: Stream to view to, e.g., stdout

6.3.5 CeedQFunction

int **`CeedQFunctionRegister`**(**`const`** char **name*, **`const`** char **source*, *CeedInt vlength*, CeedQFunctionUser *f*, int (**init*))*Ceed*, **`const`** char*, *CeedQFunction*

Register a gallery QFunction.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `name`: Name for this backend to respond to
- `source`: Absolute path to source of QFunction, "\path\CEED_DIR\gallery\folder\file.h:function_name"
- `vlength`: Vector length. Caller must ensure that number of quadrature points is a multiple of vlength.
- `f`: Function pointer to evaluate action at quadrature points. See Public API for CeedQFunction.
- `init`: Initialization function called by CeedQFunctionInit() when the QFunction is selected.

int **CeedQFunctionFieldSet**(*CeedQFunctionField *f*, **const** char **fieldname*, *CeedInt size*, *CeedEvalMode emode*)

Set a CeedQFunction field, used by CeedQFunctionAddInput/Output.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `f`: CeedQFunctionField
- `fieldname`: Name of QFunction field
- `size`: Size of QFunction field, (ncomp * dim) for *CEED_EVAL_GRAD* or (ncomp * 1) for *CEED_EVAL_NONE*, *CEED_EVAL_INTERP*, and *CEED_EVAL_WEIGHT*
- `emode`: *CEED_EVAL_NONE* to use values directly, *CEED_EVAL_INTERP* to use interpolated values, *CEED_EVAL_GRAD* to use gradients, *CEED_EVAL_WEIGHT* to use quadrature weights.

int **CeedQFunctionFieldView**(*CeedQFunctionField field*, *CeedInt fieldnumber*, bool *in*, FILE **stream*)

View a field of a CeedQFunction.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] field`: QFunction field to view
- `[in] fieldnumber`: Number of field being viewed
- `[in] in`: true for input field, false for output
- `[in] stream`: Stream to view to, e.g., stdout

int **CeedQFunctionSetFortranStatus**(*CeedQFunction qf*, bool *status*)

Set flag to determine if Fortran interface is used.

Backend Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `qf`: CeedQFunction
- `status`: Boolean value to set as Fortran status

6.3.6 CeedOperator

int **CeedOperatorCreateFallback**(*CeedOperator op*)

Duplicate a CeedOperator with a reference Ceed to fallback for advanced CeedOperator functionality.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `op`: CeedOperator to create fallback for int **CeedOperatorCheckReady**(*Ceed ceed*, *CeedOperator op*)

Check if a CeedOperator is ready to be used.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] ceed`: Ceed object for error handling
- `[in] op`: CeedOperator to check

int **CeedOperatorFieldView**(*CeedOperatorField field*, *CeedQFunctionField qffield*, *CeedInt fieldnumber*, bool *sub*, bool *in*, FILE **stream*)

View a field of a CeedOperator.

Utility Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] field`: Operator field to view
- `[in] qffield`: QFunction field (carries field name)
- `[in] fieldnumber`: Number of field being viewed
- `[in] sub`: true indicates sub-operator, which increases indentation; false for top-level operator
- `[in] in`: true for an input field; false for output field
- `[in] stream`: Stream to view to, e.g., stdout

int **CeedOperatorSingleView**(*CeedOperator op*, bool *sub*, FILE **stream*)

View a single CeedOperator.

Utility Functions

Return Error code: 0 - success, otherwise - failure

Parameters

- `[in] op`: CeedOperator to view
- `[in] sub`: Boolean flag for sub-operator
- `[in] stream`: Stream to write; typically stdout/stderr or a file

int **CeedOperatorGetActiveBasis**(*CeedOperator op*, *CeedBasis *activeBasis*)

Find the active vector basis for a CeedOperator.

@ ref Developer

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] op`: CeedOperator to find active basis for
- `[out] activeBasis`: Basis for active input vector

int **CeedOperatorMultigridLevel_Core**(*CeedOperator opFine*, *CeedVector PMultFine*, *CeedElemRestriction rstrCoarse*, *CeedBasis basisCoarse*, *CeedBasis basisCtoF*, *CeedOperator *opCoarse*, *CeedOperator *opProlong*, *CeedOperator *opRestrict*)

Common code for creating a multigrid coarse operator and level transfer operators for a CeedOperator.

Library Developer Functions

Return An error code: 0 - success, otherwise - failure

Parameters

- `[in] opFine`: Fine grid operator
- `[in] PMultFine`: L-vector multiplicity in parallel gather/scatter
- `[in] rstrCoarse`: Coarse grid restriction
- `[in] basisCoarse`: Coarse grid active vector basis
- `[in] basisCtoF`: Basis for coarse to fine interpolation
- `[out] opCoarse`: Coarse grid operator
- `[out] opProlong`: Coarse to fine operator
- `[out] opRestrict`: Fine to coarse operator

7 Developer Notes

7.1 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, ncomp, Q, nelem]
```

means that it can be traversed as

```
for (d=0; d<dim; d++)
  for (c=0; c<ncomp; c++)
    for (q=0; q<Q; q++)
      for (e=0; e<nelem; e++)
        u[((d*ncomp + c)*Q + q)*nelem + e] = ...
```

This ordering is sometimes referred to as row-major or C-style. Note that flattening such as

```
// u has shape [dim, ncomp, Q*nelem]
```

and

```
// u has shape [dim*ncomp, Q, nelem]
```

are purely implicit – one just indexes the same array using the appropriate convention.

7.2 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 `compstride` parameter. Data for node `i`, component `j`, element `k` can be found in the **L-vector** at index `offsets[i + k*elemsize] + j*compstride`.
  - **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**.

7.3 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.

7.4 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 libCEED: 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 style` 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.

8.1 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.

8.2 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.

9 libCEED Code of Conduct

9.1 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.

9.2 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

9.3 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.

9.4 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.

9.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at jed@jedbrown.org, valeria.barra@colorado.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.

9.6 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:

9.6.1 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.

9.6.2 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.

9.6.3 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.

9.6.4 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.

9.7 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.

10 Changes/Release Notes

On this page we provide a summary of the main API changes, new features and examples for each release of libCEED.

10.1 Current Main

The current `main` (formerly called `master`) branch contains bug fixes and additional features.

10.1.1 Interface changes

10.1.2 New features

- New HIP MAGMA backends for hipMAGMA library users: `/gpu/hip/magma` and `/gpu/hip/magma/det`.
- Julia and Rust interfaces added, providing a nearly 1-1 correspondence with the C interface, plus some convenience features.
- New HIP backends for improved tensor basis performance: `/gpu/hip/shared` and `/gpu/hip/gen`.
- Static libraries can be built with `make STATIC=1` and the pkg-config file is installed accordingly.

10.1.3 Performance improvements

10.1.4 Examples

- *Solid mechanics mini-app* example updated with traction boundary conditions.

10.2 v0.7 (Sep 29, 2020)

10.2.1 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 `CeedOperatorLinearAssembleAddPointBloc` 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.

10.2.2 New features

- New HIP backend: `/gpu/hip/ref`.
- CeedQFunction support for user `CUfunction`s in some backends

10.2.3 Performance improvements

- OCCA backend rebuilt to facilitate future performance enhancements.
- Petsc BPs suite improved to reduce noise due to multiple calls to `mpiexec`.

10.2.4 Examples

- *Solid mechanics mini-app* example updated with strain energy computation and more flexible boundary conditions.

10.2.5 Deprecated backends

- The `/gpu/cuda/reg` backend has been removed, with its core features moved into `/gpu/cuda/ref` and `/gpu/cuda/shared`.

10.3 v0.6 (Mar 29, 2020)

libCEED v0.6 contains numerous new features and examples, as well as expanded documentation in this new website.

10.3.1 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.

10.3.2 Performance Improvements

- MAGMA backend performance optimization and non-tensor bases.
- No-copy optimization in `CeedOperatorApply()`.

10.3.3 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()`.

10.3.4 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.

10.4 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)

10.5 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)

10.6 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)

10.7 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)

10.8 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)

10.9 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)

11 Bibliography

12 Indices and tables

- genindex
- search

References

[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.

[GRL10] F. X. Giraldo, M. Restelli, and M. Läuter. Semi-implicit formulations of the navier–stokes equations: application to nonhydrostatic atmospheric modeling. *SIAM Journal on Scientific Computing*, 32(6):3394–3425, 2010. doi:10.1137/090775889.

[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.

[PMK92] TC Papanastasiou, N Malamataris, and Ellwood K. A new outflow boundary condition. *International Journal for Numerical Methods in Fluids*, 14:587–608, 03 1992. doi:10.1002/fld.1650140506.

[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.

[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.

[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