affect Documentation

Release 0.1

Kevin Copps

May 27, 2018

Contents

CHAPTER 1

Guide to using Exodus

The `affect.exodus` module provides a Numpy array representation of the data from an Exodus database.

1.1 Calculation on all timesteps by block

Here is a relatively larger example where you want to perform a operation on every time step, on all element blocks, involving all the nodal fields in the database. Suppose your calculation is performed in the function `my_block_calculation`. Just for the sake of this example, we suppose that the result of the block calculation is a scalar, and that these are summed into a global result, which is further summed over time steps.

1.1.1 Outline

The general idea of the procedure is,

- **open the database**
    - get the node coordinates
    - **for each block**
        - get the element-to-node connectivity and keep in memory
    - **for each time step**
        - get global field values
        - **for each block**
            - gather the field values to local values on the block

- perform your calculation

Here since we want to operate on all the nodal fields block-by-block, for each time step first read in all the nodal field arrays. Following that, as we loop over blocks, we create an indexed local field array for a block.

Our strategy uses extra memory for local copies of the field values as we operate on a block, but makes the calculation efficient in terms of speed. The memory for the arrays storing the local copies of field values for one block can be garbage collected as you proceed to work on the next block.

1.1.2 Keeping local connectivities

The local connectivities (element-to-node ID connectivity for a block) are kept in compressed form in memory until the point at which they are needed. This is handled transparently by the `affect.exodus.LocalConnectivity` and using the `compress=True` option passed to `affect.exodus.Blocks.connectivity_local_all()`, emphasized below. After the connectivity is used in uncompressed form, the uncompressed copy can be garbage collected.

1.1.3 Copying fields from global to local

The local indexed copying of the nodal coordinates and other nodal field arrays are created by using the `numpy.ndarray.take()`, emphasized below.

Listing 1: block_by_block_calculation_example.py

```
with exodus.DatabaseFile('/tmp/myExodusFile.exo') as e:

    num_times = e.globals.num_times()  # read number of time steps

    nodal = e.nodal                    # get the nodal object
    fields = exodus.Fields(nodal)      # ordered dictionary of field info
    coordinates = nodal.coordinates()  # read array for all nodes

    # Read the block local connectivities and store in a dictionary.
    # And the same for node coordinates, since they don't change with
    # time step.
    local_connectivities = OrderedDict()  # maintain block order
    local_coordinates = dict()
    local_iterator = e.element_blocks.connectivity_local_all(compress=True)
    for block_id, block, local in local_iterator:
        local_connectivities[block_id] = local
        # We use the take function to select the
        # global nodes to copy into our local array
        local_coordinates[block_id] = coordinates.take(local.global_nodes,
                                                       axis=0)

    all_times_result = 0.0

    for time_step in range(num_times):  # loop over time steps

        # Read the value of all nodal field arrays on global nodes at this
```

(continues on next page)

(continued from previous page)

```
        # time step. Here you may decide to select only the subset of
        # fields you need for your calculation by name.
        global_arrays = dict()  # to hold the field arrays
        for name, field in fields:
            global_arrays[name] = nodal.field(field, time_step)  # read

        all_blocks_result = 0.0

        for block_id, local in local_connectivities:  # each block

            if local is None:
                continue  # skip blocks without any nodes

            # Copy relevant node field values from global to
            # local arrays for this block.
            local_fields = dict()
            # Add the local coordinates first, which we already have.
            local_fields['coordinate'] = local_coordinates[block_id]
            for name, array in global_arrays:
                local_fields[name] = array.take(local.global_nodes, axis=0)

            # Perform our calculation on local_fields arrays on this block.
            block_result = my_block_calculation(block_id,
                local.local_nodes, local_fields)

            all_blocks_result += block_result

        all_times_result += all_blocks_result
```

So all the arrays passed to `my_block_calculation` are in local form specific to a single block at a time.

CHAPTER 2

API Reference

The API reference provides detailed descriptions of affect's classes and functions.

2.1 `math` — Calculations on grid and mesh fields

2.1.1 `affect.arithmetic`

2.1.1.1 Usage

This module contains functions for making basic calculations on field values associated with the nodes and cells of a mesh. For example, you may want to compute the average of a node field in every cell.

It is not meant to contain every possible calculation that one would need in practice. What distinguishes this module is that it includes highly optimized and threaded functions for a few common operations.

`average_element_node_values`

2.1.1.2 Functions

2.1.1.3 Exceptions

The exceptions thrown from this module are a part of the interface just as much as the functions and classes. We define an Error root exception to allow you to insulate yourself from this API. All the exceptions raised by this module inherit from it.

2.2 connect — Connectivity Utilities

2.2.1 affect.connect

2.2.1.1 Usage

Utilities for processing connectivity information. Often the only connectivity stored on disk is the element (cell) to vertex, or element to node connectivity. The functions in this module determine element neighbor information and the vertex to element connectivity.

element_to_element
vertex_to_element
boundary_face_to_vertex
convert_to_local_connectivity

2.2.1.2 Functions

2.2.1.3 Cell and Element Topology

Standard cell to vertex connectivities are identified using the CellTopology enum.

2.2.1.4 Exceptions

The exceptions thrown from this module are a part of the interface just as much as the functions and classes. We define an Error root exception to allow you to insulate yourself from this API. All the exceptions raised by this module inherit from it.

2.3 dynamics — Structural Dynamics Analysis

2.3.1 affect.dynamics

2.3.1.1 Summary

This module contains useful functions and postprocessors concerned with analyzing the behavior of physical structures when subjected to dynamic forces. This module is useful when the applied dynamic forces result in accelerations high enough to excite the structure's natural frequency.

Dynamic analysis can be used to find dynamic displacements, time history, and modal analysis.

frf

2.3.1.2 Frequency Response Function

Frequency response is the quantitative measure of the output spectrum of a system or device in response to a stimulus, and is used to characterize the dynamics of the system. It is a measure of magnitude and phase of the output as a function of frequency, in comparison to the input. The frequency response function is a transfer function used to identify the resonant frequencies, damping and mode shapes of a physical structure.

Input Force

$$F()$$

Transfer Function

$$H()$$

Displacement Response

$$X()$$

Here, F is the input force as a function of the frequency ω, and H is the transfer function, while X is the displacement (or velocity or acceleration) response function. Each function is a complex function, with real and imaginary components, which may also be represented in terms of magnitude and phase, and thus the functions are spectral functions. For sake of computation and simplicity, we consider each to be a Fourier transform.

Thus, in the frequency domain, the structural response X(ω) is usually expressed as the product of the frequency response function H(ω) and the input or applied force F(ω). Usually the response X(ω) may be in terms of displacement, velocity, or acceleration.

$$X() = H()F()$$

$$H() = \frac{X()}{F()}$$

Using a frequency response function, the following can be observed:

- Resonances - Peaks indicate the presence of the natural frequencies of the structure under test
- Damping - Damping is proportional to the width of the peaks. The wider the peak, the heavier the damping
- Mode Shape – The amplitude and phase of multiple FRFs acquired to a common reference on a structure are used to determine the mode shape

Nomenclature:

Various transfer functions are useful for measuring system response and these have common names:

Quantity	Name of Frequency Response Function
displacement / force	admittance, compliance, receptance
velocity / force	mobility
acceleration / force	accelerance, inertance
force / displacement	dynamic stiffness
force / velocity	mechanical impedance
force / acceleration	apparent mass, dynamic mass

Example:

Examine the natural frequencies in the computational results of a structural model. Find the peak values of the *accelerance* frequency response function, where the response is acceleration in the z-direction given an input stimulus of force in the z-direction.

```
from affect.exodus import DatabaseFile
from affect.dynamics import frf
from scipy.signal import argrelmax

with DatabaseFile('./SRS-FRF-example/model/1/p1f-out.h') as e:
    times = e.globals.times()
    num_times = times.size
    node_vars = e.nodal.variables
    az = e.nodal.variable_at_times(node_vars['AccZ'], 0, 0, num_times)
    fz = e.nodal.variable_at_times(node_vars['AForceZ'], 1, 0, num_times)

frequency, h_transfer = frf(fz, az, times)
peaks = argrelmax(h_transfer)
for i, j in enumerate(peaks):
    print(i, frequency[j], h_transfer[j])
```

2.4 `exodus` — I/O of ExodusII Databases

2.4.1 `affect.exodus`

2.4.1.1 Usage and Data Model

The fundamental class for I/O is `Database`. Though it can be directly used, there is a convenient `DatabaseFile` context manager.

Access to the stored values and the structure of the `Database` is through objects of the following classes:

`DatabaseFile`
`Database`
`Global`
`Field`
`FieldArray`
`Fields`
`Nodal`
`Blocks`
`Block`
`Maps`
`Map`
`Sets`
`Set`

See also the Exodus data model *Glossary* for more information.

2.4.1.2 Array Data and Internal Memory Buffers

Wherever possible, native array data from the ExodusII C API is accessed directly without copying through a view on a `numpy.ndarray`. This maintains performance by eliminating copying, and it supplies the arrays in a convenient form for computations with numpy or scipy functions.

Some methods in this module, however, require allocating a smaller temporary memory buffers for working space. These buffers are small and the size is noted in the documentation for each method. Typical examples of the temporary memory buffer include functions required to translate Exodus C strings to the Python str type, or rearrange integer or floating point arrays in the correct order before supplying them or converting them to `numpy.ndarray`. Internal temporary buffers are allocated on the C/C++ heap using malloc or equivalent function. The buffer memory is released before the function returns. If an exception occurs, we are careful that the buffer is still released.

2.4.1.3 Database Objects

2.4.1.4 Entry Types

2.4.1.5 Exceptions and Debug Messages

The exceptions thrown from this module are a part of the interface just as much as the functions and classes. We define an Error root exception to allow you to insulate yourself from this API. All the exceptions raised by this module inherit from it.

`Error`
`NoMemory`
`FileError`
`FileExists`
`FileNotFound`
`FileAccess`
`ReadWriteError`
`InternalError`
`InvalidEntityType`
`ArrayTypeError`
`ArgumentTypeError`
`InactiveComponent`
`InactiveEntity`
`InactiveField`
`InvalidSpatialDimension`
`NotYetImplemented`
`RangeError`

See also:

example code in `Error`

2.4.1.6 Global Entity

2.4.1.7 Nodal Entity

2.4.1.8 Blocks Entities

2.4.1.9 Sets Entities

2.4.1.10 Maps Entities

2.4.1.11 Fields

2.4.1.12 Local Block Connectivity

2.4.1.13 Exceptions

2.4.1.14 Utility Functions

2.5 `util` — Utilities

2.5.1 `affect.util`

2.5.1.1 Usage

This module provides for creating and performing operations with aligned and compressed arrays.

`CompressedArray`
`empty_aligned`
`zeros_aligned`
`byte_align`
`is_byte_aligned`
`take`
`compress`
`decompress`

It also provides some basic functions for debugging and testing.

`arrays_share_data`
`get_array_base`
`ConsoleCode`
`print_blue`
`print_bold`

Continued on next page

Table 7 – continued from previous page

print_green
print_yellow
print_function_starting
print_array_info

2.5.1.2 Data alignment

Data alignment for arrays means putting the data at a memory address equal to some multiple of the word size. This is done to increase efficiency of data loads and stores to and from the processor. Processors are designed to efficiently move data to and from memory addresses that are on specific byte boundaries.

In addition to creating the data on aligned boundaries (that aligns the base pointer), the compiler is able to make optimizations when the data access (including base-pointer plus index) is known to be aligned by 64 bytes. Special SIMD instructions can be utilized by the compiler for certain platforms. For example, the compiler/platform may support the special instructions on processors such as the Intel® AVX-512 instructions, which enables processing of twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE.

By default, the compiler cannot know nor assume data is aligned inside loops without some help from the programmer. Thus, you must also inform the compiler of this alignment via a combination of pragmas (C/C++) or keywords, clauses, or attributes so that compilers can generate optimal code.

For the Intel® Many Integrated Core Architecture such (Intel® Xeon Phi™ Coprocessor), memory movement is optimal when the data starting address lies on 64 byte boundaries. Thus, by default, at least at the time of this writing it is optimal to create data objects with starting addresses that are modulo 64 bytes. For slightly less ambitious modern architectures, such as Intel® Skylake, 32 byte aligned addresses may be recommended.

2.5.1.3 Aligned arrays

These functions create and perform other operations on `numpy.ndarray` objects. All arrays created by calls to *affect*, and those used internally in *affect*, are aligned. The two main functions used to created aligned arrays are `empty_aligned()` and `zeros_aligned()` that behave similarly to `numpy.empty()` and `numpy.zeros()`, respectively.

For now this module defaults to using a 64 byte boundary. To align the data of `numpy.ndarray` to the word boundaries, during allocation it may be necessary to insert some unused bytes at the start of the block, this is data padding.

2.5.1.4 Array compression

The method of array compression is multithreaded and fast and can usually compress an array of integers to around a fourth of the original size. It does not compress arrays of floating point values as efficiently.

2.5.1.5 Printing, Debugging, Testing

Most of these functions are used internally for testing, but you may find them of value for regular use.

2.5.1.6 Exceptions

The exceptions thrown from this module are a part of the interface just as much as the functions and classes. We define an Error root exception to allow you to insulate yourself from this API. All the exceptions raised by this module inherit from it.

`Error`
`IllegalArgumentError`
`UnsupportedArrayType`

CHAPTER 3

Exodus databases

Analysis data from unstructured finite element or finite volume models are accessed through the *Exodus II* library. The `affect.exodus` module provides a Python interface for input and output of Exodus II databases.

> Exodus II is a model developed to store and retrieve data for finite element analyses. It is used for preprocessing (problem definition), postprocessing (results visualization), as well as code to code data transfer. An Exodus II data file is a random access, machine independent, binary file. The ExodusII file format and API is based on the NetCDF and HDF5 formats and API's, respectively.
>
> —EXODUS II: A Finite Element Data Model, Gregory D. Sjaardema, et al. (Documentation for ExodusII database files, including the C/C++ and FORTRAN API.)

The `affect.exodus` module maintains compact representation of the array data accessed by direct access through Numpy array objects.

See the overview and examples in the guide to Exodus to get started calling the API.

CHAPTER 4

Analysis Tools

Tools for analyzing structural dynamics are contained in `affect.dynamics`.

CHAPTER 5

Connectivity Tools

Procedures for constructing neighbor, and boundary entries for a mesh are in `affect.connect`.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

6.1 Glossary

6.1.1 Exodus Database

Conceptual data model of the ExodusII database and `affect.exodus` module.

attributes Optional floating point numbers that can be assigned to each and every entry in a `Nodal`, `Set`, or `Block` entity. Every entry in an entity must have the same number of attributes, but the attribute values vary among the entries. Attributes are accessed through the member functions of an entity, for example, `Set.num_attributes()`, `Set.attribute_names()`, `Set.attribute()`, and `Set.attributes()`.

block A association of entries with the same topology containing node connectivity information. For example, an element `Block` is an association of element entries and the nodes connected to each element.

blocks A dictionary-like container of the `Block` instances of a certain `EntityType` in a `Database`. The dictionaries of different types of blocks are accessible through the following attributes: `Database.edge_blocks`, `Database.face_blocks`, or `Database.element_blocks`.

coordinates A special field associated with the `Nodal` entity storing the spatial coordinates of every node entry in the `Database`.

database File storage for a mesh data model, a `Database` contains all the mesh entities and their corresponding entries, and the temporal `Field` variables.

distribution factors Optional floating point values associated with every entry and every `Set` of a certain type, if they exist. Distribution factors are typically used in the simulation as a multiplier on an external load. Distribution factors are accessed through `Sets.num_distribution_factors_all()`, `Set.num_distribution_factors()`, and `Set.distribution_factors()`.

entity An association of a subset of entries of a certain type (elements, faces, sides, edges, nodes). An entity is either the single `Global` or `Nodal` entity of the `Database`, or one of the possible multiple members of the `Blocks`, `Sets`, or `Maps` entities of the `Database`.

entity ID An integer associated with each existing entity in the `Database`, the integer is unique to each entity of the same `EntityType`. The entity ID's are used as the keys used to access the dictionary-like containers `Blocks`, `Sets`, and `Maps`.

entity Type One of the values of the enum `EntityType`, including `NODAL`, `NODE_SET`, `EDGE_BLOCK`, `EDGE_SET`, `FACE_BLOCK`, `FACE_SET`, `ELEM_BLOCK`, `ELEM_SET`, `SIDE_SET`, `ELEM_MAP`, `NODE_MAP`, `EDGE_MAP`, `FACE_MAP`, `GLOBAL`, and `COORDINATE`.

entry Entries are the fundamental building blocks of the grid or mesh of a database. Entries refer to nodes, edges, faces, and elements of the `Database` mesh. Entries are not represented by their own Python objects, entry IDs, but they correspond to the first index of the `FieldArrays`.

field A name for an array of values and the name of components associated with entries. The `Field` names are used to access the `FieldArray` values stored in the `Database`. Each of the named components of a `Field` with values in a `FieldArray` are a scalar *variable* in the `Database`. A *field* is a grouping of ExodusII *variable* by common name prefix; the suffix of the *variable* name (whatever follows the last underscore '_' in the name) becomes a component name of the field. See also *field array*.

field array The actual scalar, vector and tensor values accessed in the `Database` by using a `Field` name and components. The `FieldArray` is a multidimensional array, with the first index corresponding to entries. It contains floating point values that vary in space (by *entry* index) and time (*time step*). Entities that may have field array values: global, nodal, blocks, and sets. For fields on blocks or sets, the field may or may not be active on all entities of that type; to find out use `Block.is_field_active()` or `Set.is_field_active()`. The values of the field array may be accessed on all entries at a single time step, for example see `Nodal.field()`; or on a range of entries at a time step, for example, `Nodal.partial_field()`; or on a single entry at all existing time steps, for example, `Nodal.field_at_times()`.

global A `Global` is a single top level `Database` entity maintaining the spatial dimension, the number of time steps, the sums of all the entries of various types in the mesh (elements, faces, nodes) referenced in other `Database` entities. It is accessed from the attribute `Database.global`.

information data Info data is a list of optional supplementary text strings associated with a database. Typically this might be the input file from the simulation run that was executed to create the database results. Information data is accessed through `Database.info`

internal numbering The internal numbering of node entries is in the range [0, `Global.num_nodes()`]. The internal numbering of elements is by total subsequent entries in the `Block` in `Database.`

`blocks()` (of type `EntityType.ELEMENT_BLOCK`) and these are in the range [0, `Global.num_elements()`].

map A `Map` is a container of entries with new integers representing a number other than that of the default internal numbering for that type of entry.

maps A dictionary-like container of the `Map` instances of a certain `EntityType` in a `Database`. The dictionaries of different types of maps are accessible through the following attributes: `Database.element_maps`, `Database.node_maps`, `Database.edge_maps`, or `Database.face_maps`.

quality assurance records QA data are optional text strings in the `Database`, storing a history of application codes that modified the `Database` file, including the application name, description, date and time. Quality assurance data is accessed through `Database.qa_records`

nodal The single entity of a `Database` that stores nodal coordinates, nodal fields, and nodal attributes. The `Nodal` object is accessed from `Database.nodal`.

properties Optional named integer variables associated with every entity of a certain type in the database. The types of entities that may have properties are: `Block`, `Map`, and `Set` entities. Property names are accessed through the member function of the collection of entities, for example, `Blocks.property_names()`. Property values are accessed through the member functions of an entity, for example, `Block.property()`.

set A `Set` entity is a container of a subset of the entries of a certain type (nodes, edges, faces, sides, elements) in the `Database`. There may be multiple sets of a certain type and they may intersect. Sets are usually used to apply boundary conditions to portions of the mesh, and sets may contain *attributes*, *properties* and *distribution factors*, and multiple *variable*.

sets A dictionary-like container of the `Set` instances of a certain `EntityType` in a `Database`. The dictionaries of different types of sets are accessible through the following attributes: `Database.node_sets`, `Database.edge_sets`, `Database.face_sets`, or `Database.side_sets`. Entries of side sets are actually the pairing of an element and a local side number.

variable Variables, in a `Database` are named scalar floating point arrays. The values of variables vary in time and are associated with entries in the database. A single variable is one component of a more useful multi-dimensional `FieldArray`, there is often no need to refer to variables separately from a `FieldArray`. The suffix of a name of a Exodus variable is also the name of a `Field` component. The underlying scalar variable values making up *field array* may be accessed in the database in a similar way to their `FieldArray` counterpart.

time step The discrete values of time at which the values of fields (variables) are stored in the database. The values of time steps are accessible through the attribute `Database.globals.num_times` and `Database.globals.times`.

Index