Part 4: Software Engineering - Wave Equation Model

Code Optimization Techniques

Petar Bosnic

University of South-Eastern Norway

November 27, 2024

Note

This document was developed as of the PhD course *Numerical Solutions to Partial Differential Equations* under the guidance of Professors Svein Linge and Knut Vågsæther at the University of South-Eastern Norway (USN). Course material includes the textbooks:

- *Finite Difference Computing with PDEs: A Modern Software Approach* by Hans Petter Langtangen and Svein Linge (DOI: 10.1007/978-3-319-55456-3)
- *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction* by E. F. Toro (ISBN: 978-3-540-25202-3 978-3-540-49834-6)

Contents

1 Saving Large Arrays in Files

1.1 Saving Large Arrays in Files Using `numpy.savez`

When performing numerical simulations, large arrays often need to be saved to disk for later analysis.NumPy provides a more efficient and optimized method through `numpy.savez`. This function allows multiple arrays to be saved in a single compressed `.npz` file, using meaningful, user-defined names.

1.1.1 Dynamic Naming for Arrays

In simulations, arrays are saved at different time steps, so dynamic naming is used to reflect the specific time step in the file or array name. For example, if `u` and `v` are arrays at time step `n`, their names can be dynamically generated using Python string formatting:

```
n = 11 # Example time step
u_name = 'u%04d' % n # Generates 'u0011'
v_name = 'v%04d' % n # Generates 'v0011'
```

Here, `%04d` ensures that the integer `n` is zero-padded to four digits, creating consistent names like `u0011` or `v0011`.

1.1.2 Creating a Dictionary of Arrays

The arrays to be saved are added to a dictionary where the keys are the dynamic names (e.g., `'u0011'`, `'v0011'`) and the values are the corresponding array data:

```
import numpy as np

# Example arrays
u = np.random.rand(100) # Simulated data for u
v = np.random.rand(100) # Simulated data for v

kwargs = {u_name: u, v_name: v} # Create a dictionary with names and
    arrays
```

1.1.3 Saving Arrays Using `numpy.savez`

The `numpy.savez` function saves multiple arrays into a single compressed `.npz` file. In this method, a dictionary is created where the keys are dynamically generated names (e.g., `'u0011'`, `'v0011'`) that include the time step index `n`, and the values are the corresponding array data (e.g., `u`, `v`). This dictionary is then passed to `numpy.savez` using the unpacking operator `**kwargs`, which converts the dictionary into keyword arguments for the function. A dynamic file name is also created to reflect the current time step, ensuring systematic file organization.

```
# Create dynamic file name and array names
fname = '.mydata%04d.dat' % n # Example: '.mydata0011.dat'
u_name = 'u%04d' % n        # Example: 'u0011'
v_name = 'v%04d' % n        # Example: 'v0011'

# Create a dictionary with dynamic names and arrays
kwargs = {u_name: u, v_name: v}
```

```
# Save arrays to a compressed .npz file
np.savez(fname, **kwargs)
```

In this example:

- `%04d` formats the integer `n` to zero-pad it to four digits, ensuring consistency in names and file organization.
- The dictionary `kwargs` contains key-value pairs where:
  - **Keys** (e.g., `'u0011'`, `'v0011'`) are the dynamically generated names for the arrays.
  - **Values** are the actual arrays to be saved.
- The `**kwargs` operator unpacks the dictionary, passing each key-value pair as a keyword argument to `numpy.savez`.

For instance, if `kwargs` is:

```
kwargs = {'u0011': u, 'v0011': v}
```

The `np.savez` call is equivalent to:

```
np.savez('.mydata0011.dat', u0011=u, v0011=v)
```

This approach automates the process, making it both flexible and efficient. It avoids repetitive manual specification of names and arrays while ensuring data is stored systematically with meaningful names for easy retrieval later.

Structure: `kwargs` is simply a dictionary where:

- **Keys** represent the argument names (e.g., `'u0011'`, `'v0011'`).
- **Values** are the actual data associated with those names (e.g., the arrays `u` and `v`).

Unpacking: The `**` operator unpacks the dictionary so that its key-value pairs are sent to the function as named arguments.

```
kwargs = {'u0011': u, 'v0011': v}
np.savez(fname, **kwargs) # Equivalent to np.savez(fname, u0011=u, v0011
    =v)
```

This technique is particularly useful when the number of arguments is variable or when argument names need to be generated dynamically.

1.1.4 Saving Static Metadata

In many simulations, metadata such as spatial grid points (`x`) remains constant and does not need to be saved repeatedly for each time step. Such data can be saved once in a separate file:

```
if n == 0: # Save metadata only at the first time step
    x = np.linspace(0, 1, 100) # Example grid points
    np.savez('.mydata_x.dat', x=x) # Save metadata to a separate file
```

1.1.5 Loading the Data

The stored `.npz` file can later be loaded using `numpy.load`, and individual arrays can be accessed by their names:

```
data = np.load('.mydata0011.dat') # Load the saved file
u_loaded = data['u0011'] # Access the array u at time step 11
v_loaded = data['v0011'] # Access the array v at time step 11
```

1.1.6 Merging and Reading Zip Archives in NumPy

In numerical simulations, individual calls to `numpy.savez` produce separate `.npz` files for each dataset. To streamline file management and improve convenience, these archives can be merged into a single zip archive. The function `merge_zip_archives` achieves this by combining multiple `.npz` files into a single archive. After merging, the original `.npz` files are deleted, leaving only the combined archive.

Function Definition and Inputs The function accepts two arguments:

- `individual_archives`: A list of `.npz` file names or a wildcard pattern (e.g., `'*.npz'`) for selecting files with `glob.glob`.
- `archive_name`: The name of the resulting merged archive.

```
def merge_zip_archives(individual_archives, archive_name):
    """Merge individual zip archives made with numpy.savez into one archive."""
    import zipfile
    import glob
    import os
```

Steps in the Function :

Handle Input Files: The function first checks if `individual_archives` is a list/tuple of file names or a string. If it is a string, `glob.glob` is used to generate a list of matching file names:

```
if isinstance(individual_archives, (list, tuple)):
    filenames = individual_archives
elif isinstance(individual_archives, str):
    filenames = glob.glob(individual_archives)
```

Create the Final Archive: A new zip archive is opened in write mode:

```
archive = zipfile.ZipFile(archive_name, 'w', zipfile.ZIP_DEFLATED, allowZip64=True)
```

Merge Individual Archives: Each `.npz` file is opened, its contents extracted, and added to the new combined archive. The `.npy` extension is removed for cleaner naming:

```
for filename in filenames:
    f = zipfile.ZipFile(filename, 'r', zipfile.ZIP_DEFLATED) # Open each archive
```

```python
            for name in f.namelist(): # List all files (arrays) in the
                archive
                data = f.open(name, 'r') # Open each file
                archive.writestr(name[:-4], data.read()) # Write to new
                    archive (remove .npy)
            f.close()
            os.remove(filename) # Delete the original archive
```

Close the Archive: Once all files have been added, the archive is closed:

```python
        archive.close()
```

Benefits of the Merging Process

- **Simplifies File Management:** Combines multiple files into one, making it easier to store, or back up data.
- **Removes Redundancy:** Deletes the original `.npz` files after merging.

Usage Example To merge all `.npz` files in the current directory into a single archive named `merged_archive.zip`:

```python
merge_zip_archives('*.npz', 'merged_archive.zip')
```

1.2 Using `joblib` to Store Arrays in Files

The `Storage` class described below simplifies saving and retrieving data objects by name, wrapping `joblib`'s functionality.

1.2.1 Overview of the `Storage` Class

The `Storage` class is designed to:

- Save Python data structures (e.g., arrays, dictionaries) to disk with a specific name.
- Retrieve previously saved data objects using their names.
- Use a user-specified directory to store data, enabling flexible organization.

The class leverages the `joblib.Memory` object and Python's concept of memoization, which caches results of function calls for efficient reuse.

1.2.2 How the `Storage` Class Works

1. Initialization (`__init__` Method) The class initializes with the following parameters:

- `cachedir`: Specifies the directory where objects are stored.
- `verbose`: Controls verbosity during save and retrieve operations.

```python
def __init__(self, cachedir='tmp', verbose=1):
    """Initialize the storage system."""
    import joblib
    self.memory = joblib.Memory(cachedir=cachedir, verbose=verbose) # Set up caching
    self.verbose = verbose
    self.retrieve = self.memory.cache(self.retrieve, ignore=['data']) # Caching logic
    self.save = self.retrieve # Save operation reuses retrieve logic
```

Key points:

- `joblib.Memory` manages the caching mechanism, storing data in the directory specified by `cachedir`.
- `self.retrieve` is wrapped with `self.memory.cache`, enabling memoization for the retrieve function.

2. The `save` and `retrieve` Methods Both `save` and `retrieve` operations rely on the `retrieve` function. The caching mechanism ensures:

- If the data with the specified name has not been stored before, it is saved to disk.
- If the data with the specified name already exists, it is fetched directly from disk.

```python
def retrieve(self, name, data=None):
    if self.verbose > 0: # Print info if verbosity is enabled
        print('joblib save of', name)
    return data
```

1.2.3 Memoization and Caching

Memoization stores the results of function calls so repeated calls with the same arguments can return cached results. In this context:

- `self.memory.cache(self.retrieve, ignore=['data'])` wraps the `retrieve` function with caching logic.
- The `name` parameter determines whether to cache, save, or retrieve the object.
- Large data objects do not need to be recreated or re-saved repeatedly.

1.3 Using a Hash to Create a File or Directory Name

In simulations, organizing and storing results systematically is essential. A reliable method is to use a hash string, which encodes input data into a unique and concise file or directory name.

A hash string is a identifier generated from input data. It uniquely represents the input. Hash functions like SHA1 (Secure Hash Algorithm 1) produce 40-character-long strings and are widely used in systems like Git to uniquely identify files or changes.

1.3.1 Implementation Example

The following function generates a hash string from various input types:

```python
import inspect # For extracting source code of functions
import joblib # Efficient hashing for arrays
import hashlib # SHA1 hashing

def generate_hash(func1, func2, array1, array2, obj1, obj2):
    """Generate a hash string based on input data."""
    # Convert inputs into a tuple of strings/hashable data
    data = (
        inspect.getsource(func1), # Source code of func1
        inspect.getsource(func2), # Source code of func2
        joblib.hash(array1),     # Hash of array1
        joblib.hash(array2),     # Hash of array2
        str(obj1),               # String representation of obj1
        str(obj2)                # String representation of obj2
        )
    # Generate an SHA1 hash from the combined data
    hash_input = hashlib.sha1(str(data).encode('utf-8')).hexdigest()
    return hash_input
```

Step-by-Step Explanation

1. **Input Data Processing:**
    - **Functions:** Use `inspect.getsource(func)` to retrieve source code.
    - **Arrays:** Use `joblib.hash` for hashing.
    - **Objects:** Use `str()` to convert objects into string representations.
2. **Hash Creation:** Combine processed input data into a tuple, then hash the tuple using `hashlib.sha1`.

1.3.2 Example Usage

Given the following inputs:

```python
x0 = 0.5
func1 = lambda x: x**2
func2 = lambda x: 0 if x <= x0 else 1

import numpy as np
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
obj1 = {'key': 'value'}
obj2 = [7, 8, 9]
```

Generate a hash:

```python
hash_string = generate_hash(func1, func2, array1, array2, obj1, obj2)
print("Generated Hash:", hash_string)
```

Output:

Generated Hash: b6abd18caa7319e0a46797fd5dbaaf737e47fc64

2 Programming with Classes

Using object-oriented programming (OOP), we can design software around classes that combine data (attributes) and behavior (methods). Unlike function-based solvers, where you have to pass all input data as arguments, OOP allows us to combine related functionality into classes, making the code cleaner and easier to work with. The class structure/OOP approach used for developing the Wave Equation solver (full code can be found: wave1D_oo.py), is summarized.

2.1 Class implementation for solving of the wave equation

2.1.1 Parameters Class

The `Parameters` class serves as a structured framework for managing simulation parameters dynamic manner. Key features include:

- **Parameter Validation:** Ensures all parameters, types, and descriptions are correctly defined.
- **Dynamic Parameter Management:** Allows adding, updating, and retrieving parameters via method calls or dictionary-like syntax.
- **Command-line Integration:** Supports parameter definitions via command-line arguments using Python's `argparse`.

The class provides several methods to streamline parameter handling:

- `__init__`: Initializes three dictionaries:
  - `prm`: Stores parameter names and default values.
  - `type`: Specifies the data type of each parameter.
  - `help`: Stores descriptive text for each parameter.

  These dictionaries must be defined in subclasses of `Parameters`.
- `ok`: Validates that the `prm`, `type`, and `help` dictionaries exist and are properly structured, raising an error if they are incomplete or misconfigured.
- `set(**parameters)`: Updates the values of one or more parameters, validating their names before setting values.
- `get(name)`: Retrieves the value of a single parameter or a list of parameters, validating their existence.
- `__getitem__` and `__setitem__`: Enable dictionary-style access to parameters:
  - `obj[name]` fetches a parameter value (equivalent to `get(name)`).
  - `obj[name] = value` updates a parameter value (equivalent to `set(name=value)`).
- `define_command_line_options(parser=None)`: Extends an `argparse.ArgumentParser` object with options based on the defined parameters. It enforces data types using `self.type` and descriptive text using `self.help`.

- `init_from_command_line(args)`: Initializes parameters from parsed command-line arguments, updating `prm` values based on `args`.

2.1.2 Problem Class

This section introduces the `Problem` class, which specializes the `Parameters` class to define and solve a one-dimensional wave equation. The class defines problem-specific parameters. Below is a detailed overview:

Attributes The class is initialized as following:

```
                                def __init__(self):
                                        self.prm = dict(L=2.5, c=1.5, T=18)
                                        self.type = dict(L=float, c=float, T=float)
                                        self.help = dict(
                                        L='1D domain',
                                        c='coefficient (wave velocity) in PDE',
                                        T='end time of simulation'
                                        )
```

- `prm`: Stores parameters such as:
  - `L`: Domain length.
  - `c`: Wave velocity.
  - `T`: Simulation end time.
- `type`: Specifies parameter types (floats for `L`, `c`, and `T`).
- `help`: Provides descriptions for parameters

Methods

- `u_exact(self, x, t)`: Computes the exact solution:

$$u(x,t) = x(L-x)(1+t^2).$$

- `I(self, x)`: Computes the initial condition:

$$I(x) = u(x,0) = x(L-x).$$

- `V(self, x)`: Computes the initial velocity:

$$V(x) = 0.5 \cdot u(x,0) = 0.5 \cdot x(L-x).$$

- `f(self, x, t)`: Computes the source term:

$$f(x,t) = 2(1+0.5t)c^2.$$

- `U_0(self, t)`: Defines the boundary condition at $x=0$:

$$U_0(t) = u(0,t) = 0.$$

- `U_L(self, t)`: Represents the boundary condition at $x=L$, which is set to `None` by default.

2.1.3 Mesh Class

This section outlines the `Mesh` class, which provides a structured framework for constructing spatial and temporal grids in numerical simulations. Below is a detailed description of its purpose, attributes, and methods.

Purpose The `Mesh` class is designed to:

- Generate spatial and temporal grids for numerical simulations.
- Allow flexibility by enabling users to specify either the number of divisions (`N`) or the resolution (`d`) for grids.

Constructor: `__init__` The constructor initializes the spatial and temporal grids based on the input parameters:

- **Spatial Grid:** Users can specify either `N` (number of divisions) or `d` (grid spacing), while `L` (domain bounds) is always required.
- **Temporal Grid:** Users can specify either `Nt` (number of time steps) or `dt` (time step size), while `T` (total simulation time) is always required.

Key Methods

- `get_num_space_dim(self)`:
  - Returns the number of spatial dimensions.
  - If the spatial mesh is undefined (`self.d` is `None`), returns 0.
- `has_space(self)`:
  - Returns `True` if a spatial mesh is defined, `False` otherwise.
- `has_time(self)`:
  - Returns `True` if a temporal mesh is defined, `False` otherwise.
- `dump(self)`:
  - Generates a summary string describing the mesh configuration for both space and time.
  - Includes details about domain bounds, grid spacing, number of divisions, time interval, and time steps.

2.1.4 Function Class

This section introduces the `Function` class, designed to store and manage function values over a discretized domain.

Purpose The `Function` class:

- Stores these values in an appropriately shaped array (`u`) based on the `Mesh` configuration.
- Supports single-component (scalar) or multi-component (vector) functions.

Constructor: `__init__` Inputs:

- `mesh`: An instance of the `Mesh` class defining the spatial and temporal discretization.
- `num_comp`: Number of components in the function:
  - 1: For scalar functions.
  - Greater than 1: For vector or multi-component functions.
- `space_only`: Determines if the function is defined only on the spatial mesh (`True`) or on both spatial and temporal meshes (`False`).

Key Code Blocks Space-Only Mesh:

```
if (self.mesh.has_space() and not self.mesh.has_time()) or \
    (self.mesh.has_space() and self.mesh.has_time() and \
    space_only):
    if num_comp == 1:
        self.u = np.zeros(
        [self.mesh.N[i] + 1 for i in range(len(self.mesh.N))])
        self.indices = ['x'+str(i) for i in range(len(self.mesh.N))]
    else:
        self.u = np.zeros(
        [self.mesh.N[i] + 1 for i in range(len(self.mesh.N))] +
        [num_comp])
        self.indices = ['x'+str(i) for i in range(len(self.mesh.N))] +\
    ['component']
```

Time-Only Mesh:

```
if not self.mesh.has_space() and self.mesh.has_time():
    if num_comp == 1:
        self.u = np.zeros(self.mesh.Nt+1)
        self.indices = ['time']
    else:
        self.u = np.zeros((self.mesh.Nt+1, num_comp))
        self.indices = ['time', 'component']
```

Spatiotemporal Mesh:

```
if self.mesh.has_space() and self.mesh.has_time() \
and not space_only:
    size = [self.mesh.Nt+1] + \
    [self.mesh.N[i]+1 for i in range(len(self.mesh.N))]
    if num_comp > 1:
        self.indices = ['time'] + \
        ['x'+str(i) for i in range(len(self.mesh.N))] +\
        ['component']
```

```
                    size += [num_comp]
                else:
                    self.indices = ['time'] + ['x'+str(i)
                    for i in range(len(self.mesh.N))]
                self.u = np.zeros(size)
```

2.1.5 Solver Class

The `Solver` class is a implementation for numerically solving the wave equation:

$$u_{tt} = (c^2 u_x)_x + f(x,t), \quad t \in [0,T],\ x \in [0,L].$$

Below is a detailed breakdown of its functionality.

Purpose The `Solver` class:

- Numerically solves the wave equation using finite differences in time and space.
- Exploits symmetry to reduce the computational domain to $x \in [0, L/2]$, improving efficiency.

Initialization **Constructor: `__init__`**

- **Inputs:**
  - `problem`: An instance of the `Problem` class defining the wave equation parameters and exact solution.
- **Key Attributes:**
  - `C`: Courant number, influencing stability and accuracy.
  - `Nx`: Number of spatial mesh points.
  - `stability_safety_factor`: Safety factor for stability constraints.
- **Mesh and Function Setup:**
  - Defines a spatial and temporal mesh using the `Mesh` class.
  - Reduces the computational domain to $[0, L/2]$ using symmetry.
  - Computes the time step (Δt) based on the Courant condition:

$$\Delta t = \frac{\Delta x \cdot \text{stability\_safety\_factor} \cdot C}{c}.$$

  - Initializes a `Function` object (`self.f`) to store the solution over the mesh.

Solver Method: `solve()`

- **Inputs:**
  - `user_action`: Optional callback function for custom actions during the simulation.
  - `version`: Determines whether scalar (`'scalar'`) or vectorized (`'vectorized'`) computations are used.
- **Steps:**
  1. **Initialization:**
     - Extracts parameters such as L, c, T, and $f(x, t)$ from the `Problem` instance.
     - Sets up spatial and temporal grids:
       * Spatial grid: $x \in [0, L/2]$.
       * Temporal grid: $t \in [0, T]$.
     - Handles $c(x)$ as a constant, callable function, or array.
  2. **Handle Initial and Boundary Conditions:**
     - Wraps user-defined functions (`f`, `I`, `V`, `U_0`, `U_L`) to provide default behaviors.
     - Loads initial conditions $u(x, 0) = I(x)$ into the first time step.
  3. **Time-Stepping Loop:**
     - **First Time Step:**
       * Computes $u(x, t_1)$ using a formula incorporating the initial velocity $V(x)$.
       * Applies boundary conditions at $x = 0$ and $x = L/2$.
     - **Subsequent Time Steps:**
       * Updates the solution using the finite-difference scheme:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \left(\frac{2q_{i+1} + q_i}{2} (u_{i+1}^n - u_i^n) - \frac{2q_{i-1} + q_i}{2} (u_i^n - u_{i-1}^n) \right) + \Delta t^2 f(x_i, t_n).$$

       * Applies boundary conditions at each time step.
  4. **User-Defined Actions:**
     - Calls `user_action(u, x, t, n)` at each time step, if provided.
     - Allows users to visualize, analyze, or manipulate the solution during the simulation.
  5. **Efficiency Features:**
     - Supports scalar and vectorized computations for better performance.
     - Uses a hashed input file (`.npz`) to check if a simulation with identical parameters has already been run.

2.1.6 Function: `test_quadratic_with_classes()`

Function Overview

1. **Define Problem and Solver Instances**

```python
problem = Problem()
solver = Solver(problem)
```

- The `Problem` class defines the wave equation parameters and exact solution.
- The `Solver` class handles numerical computations over the reduced domain $[0, L/2]$.

2. **Define Command-Line Options**

```python
parser = problem.define_command_line_options()
parser = solver.define_command_line_options(parser)
args = parser.parse_args()
```

- Combines options from `Problem` and `Solver`.
- Allows users to specify parameters such as domain length (`L`) and time duration (`T`).

3. **Initialize Problem and Solver**

```python
problem.init_from_command_line(args)
solver.init_from_command_line(args)
```

Updates `Problem` and `Solver` attributes based on user inputs.

4. **Print Parameters for Verification**

```python
print(parser.parse_args())
```

Confirms that simulation parameters are correctly initialized.

5. **Solve the Wave Equation**

```python
solver.solve()
```

- Sets up the mesh, initial conditions, and time-stepping scheme.
- Computes the numerical solution stored in `solver.f.u`.

6. **Validate Numerical Solution**

```python
print('Check error.........................')
solver.assert_no_error()
```

- Compares the numerical solution with the exact quadratic solution.
- Ensures the error is below a tolerance (e.g., 10^{-13}).

7. **Run the Test**

```python
if __name__ == '__main__':
    test_quadratic_with_classes()
```

Purpose: Executes the `test_quadratic_with_classes()` function when the script is run directly.

Acknowledgments

I acknowledge the use of ChatGPT for writing assistance and code debugging in the preparation of this document.