Advances in Engineering Software 92 (2016) 27–39

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

SMT: An interface for localized storm surge modeling

Tristan Dyer, John Baugh*

Department of Civil, Construction, and Environmental Engineering, North Carolina State University, Raleigh, NC 27695, United States

ARTICLE INFO

Article history:
Received 20 April 2015
Revised 24 September 2015
Accepted 12 October 2015

Keywords:
Hurricane storm surge
Subdomain modeling
Finite element analysis
ADCIRC
Range search
Visualization

ABSTRACT

The devastation wrought by Hurricanes Katrina (2005), Ike (2008), and Sandy (2012) in recent years continues to underscore the need for better prediction and preparation in the face of storm surge and rising sea levels. Simulations of coastal flooding using physically based hydrodynamic codes like ADCIRC, while very accurate, are also computationally expensive, making them impractical for iterative design scenarios that seek to evaluate a range of countermeasures and possible failure points. We present a graphical user interface that supports local analysis of engineering design alternatives based on an exact reanalysis technique called *subdomain modeling*, an approach that substantially reduces the computational effort required. This interface, called the Subdomain Modeling Tool (SMT), streamlines the pre- and post-processing requirements of subdomain modeling by allowing modelers to extract regions of interest interactively and by organizing project data on the file system. Software design and implementation issues that make the approach practical, such as a novel range search algorithm, are presented. Descriptions of the overall methodology, software architecture, and performance results are given, along with a case study demonstrating its use.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Hurricane storm surge poses a threat to coastal communities, which risk destruction to life and property. The succession of events that begins with wind and pressure fields and results in catastrophic, overland flooding is mathematically complex and operates over multiple scales of time and space. While the aftereffects of coastal storms are readily apparent, predicting them remains a challenge because of the computational effort demanded by large-scale storm surge models. Multiplying the challenge, from an engineering perspective, is the need to consider – and simulate surge events over – alternative topographies that represent hypothetical design and failure scenarios of levees and other critical infrastructure.

To make storm surge simulations more practical for engineering analysis, we have developed an approach, called *subdomain modeling* [1], that substantially reduces the computational effort required to evaluate multiple topographic changes in a geographic region of interest. Using this approach, a storm event is first simulated on a large-scale domain, such as the western North Atlantic Ocean, that is able to capture the evolution of processes that originate far from the engineer's region of interest [2]. Then, design and failure scenarios are introduced on a local domain that makes use of boundary conditions obtained from the full domain, thereby avoiding computations that would fall outside the sphere of influence of any changes that have been introduced. The intuition is that the local failure of a levee or rebuilding of a stretch of dunes, say, remains local, and is unlikely to create hydrodynamic effects tens of miles up the coastline.

Because subdomain modeling suggests new steps in the simulation workflow, we introduce a graphical user interface that allows modelers to work directly with large-scale grids, extract regions of interest, and manage projects and files for simulation and analysis. We begin with background on ADCIRC and subdomain modeling, and continue with motivation, design, and implementation considerations.

1.1. ADCIRC and subdomain modeling

The Advanced Circulation model (ADCIRC) is a parallel, unstructured-grid finite element hydrodynamic code that supports three dimensional (3D) and two dimensional depth integrated (2DDI) analyses [3]. It is used by the U.S. Army Corps of Engineers (USACE), the Federal Emergency Management Agency (FEMA), and others to simulate storm surge and tides along the East coast of the United States and elsewhere [4,5]. The 2DDI formulation used in subdomain modeling is derived from the vertically integrated shallow water equations using the generalized wave continuity equation (GWCE) formulation [3].

Three primary routines drive the physics of the ADCIRC model and are executed in succession at each timestep: the GWCE routine, the wet/dry routine, and the momentum equation routine [4]. The GWCE routine, which determines the free surface elevation at each node in the domain for the current timestep, can use either a spatially implicit

* Corresponding author. Tel.: +1 919 515 7697.
E-mail address: jwb@ncsu.edu (J. Baugh).

http://dx.doi.org/10.1016/j.advengsoft.2015.10.003
0965-9978/© 2015 Elsevier Ltd. All rights reserved.

or explicit method, employing an iterative Jacobi Conjugate Gradient method or a lumped mass matrix method, respectively. Subdomain modeling uses the implicit method. The wet/dry routine determines the wet or dry status of each node based on elevation, velocity, wet/dry status in the previous timestep, and wet/dry status of neighboring nodes [6]. The momentum routine determines the *x* and *y* velocity at each node for the current timestep by explicitly solving the shallow water equations using a lumped mass matrix.

The necessity and effectiveness of large-scale approaches for tide and storm surge modeling are addressed by Blain et al. [2], who examine combinations of domain size and boundary conditions for their effects on computed storm surge characteristics. The authors conclude that small domains, whose boundaries are in the same regions where surge effects appear, are inadequate because the surge effects on those boundaries cannot be known, and hence enforced, a priori. The open ocean boundaries of very large domains, however, are far enough removed from these regions that the influence of the boundary condition specification is minimal, and such domains are found to be the most practical and effective.

In order to retain the accuracy and effectiveness of large-scale simulations in ADCIRC while providing the speed and portability of small-scale simulations, we have developed the subdomain modeling approach. Subdomain modeling allows one to perform a series of hurricane simulations on multiple, alternative topographies in a geographic region of interest. This capability is achieved by recording data at the boundaries of the region of interest during the simulation of a storm event over the entire geographic region (called a full domain). The recorded data are then used to enforce the boundary values during a subdomain run using a new type of boundary condition. Like the domain decomposition solver used in Parallel ADCIRC [4], the new boundary condition makes use of water surface elevation, wet/dry status, and depth averaged water velocity by:

- working with the existing non-periodic elevation boundary condition formulation in ADCIRC, which specifies nodal elevations in the implicit GWCE formulation,
- incorporating the ability to force wet/dry status on boundary nodes while the wetting and drying routine executes, and
- taking advantage of the explicit nature of the momentum equation solver to assign boundary velocities outright.

By employing this new boundary condition, a subdomain can produce a solution that matches a full domain run in a small fraction of the time. Modifications to ADCIRC enable this basic functionality and are now part of the official distribution [7]; those implementation details are provided in a companion paper [1]. The focus here is on a graphical user interface, the Subdomain Modeling Tool (SMT), that improves upon the command-line interface by providing an intuitive and interactive workflow for subdomain modeling, while offering compatibility with existing mesh development tools in an extensible architecture designed for cross-platform expansion and future growth.

2. Background, functionality, and design goals

Before laying out design goals for a more sophisticated user interface, we begin with the underlying building blocks on which such an interface could be built. These low-level steps are largely driven by the mathematical formulation of coastal hydrodynamic codes like ADCIRC, and by the computational aspects of subdomain modeling.

The construction of a subdomain model consists of four main steps, as outlined in Fig. 1. Initially, a region of interest is identified, and subdomain input and control files are generated. Then, a full domain ADCIRC run is performed, after which a subdomain boundary conditions file is extracted from the full domain output files. Finally, a subdomain ADCIRC run is performed, generating results that can be used to verify the subdomain model against the full domain.

Following the construction of the subdomain model, subsequent case studies can be carried out by iteratively making local changes to the subdomain mesh and performing an ADCIRC run on the modified subdomain. After each iteration, the modified subdomain can be verified to ensure that the effects of the changes do not propagate to the subdomain boundaries, or that the extent of the variations at the boundaries are within an acceptable tolerance, at the discretion of the modeler. Additional details and case studies describing the verification of subdomain boundaries and results are presented elsewhere [1,8].

Considering a couple of these steps in more detail, in the first, a region of interest can be identified using simple geometric shapes with numerical parameters given textually. Absent visual interaction, the modeler confirms by some other means that the extracted area matches her region of interest. This step produces the required ADCIRC input files, along with additional ones for subdomain modeling, though manual intervention is needed for adjusting some of ADCIRC's control parameters. In the third step, the modeler can generate the subdomain boundary conditions files using the basic building blocks provided. The generated files must then be placed in expected locations before moving on to step four and performing a subdomain run. The verification of the subdomain model can be performed by generating images and hydrographs at specific nodal locations, and by manually comparing their values.

In practice, studies employing subdomain modeling lead to multiple, alternative subdomain meshes. For example, the design of a levee system could use a single region of interest to test a variety of levee configurations and failure scenarios. Upon creating a subdomain model, each levee configuration would be incorporated into a unique copy of the subdomain mesh, and each of these unique configurations would be generated by the modeler performing the case study. Getting new topographic features into a mesh might be achieved either with SMT, or by using other software packages developed specifically for the purpose of re-meshing and mesh editing. One of the more popular tools used within the ADCIRC community and by other surface-water modelers is SMS (Surface-water Modeling System), a proprietary software product developed and maintained by Aquaveo [9]. SMS provides interactive tools for generating and editing ADCIRC meshes, although they do not include the features necessary to create subdomains or manage the file formats specific to subdomain modeling.

2.1. Desired functionality

Building on these basic capabilities, we outline the design goals of a more interactive and visually oriented interface for subdomain modeling that also includes project management features.

- **Targeting regions of interest** - A visual representation of ADCIRC meshes, capable of displaying topographies and nodal attributes with customizable color gradients, allows modelers to identify features and regions of interest interactively by panning and zooming. Selection tools would then allow the modeler both to select and deselect elements to be included in the subdomain. Ideally, interaction with a mesh would be rapid and direct. Based on empirical studies, operations that are able to complete within about 100 ms appear as though immediate [10,11].
- **Managing subdomains** - An ability to create, manage, and view the location of files is needed for a streamlined workflow. Files specific to subdomain modeling, such as those enforcing boundary conditions, would be created automatically at the appropriate stage within the workflow.
- **Subdomain verification** - Following the construction of a subdomain model, the subdomain grid should be verified to ensure that results of the subdomain and full domain runs match. Support for this verification step would include visual and numerical

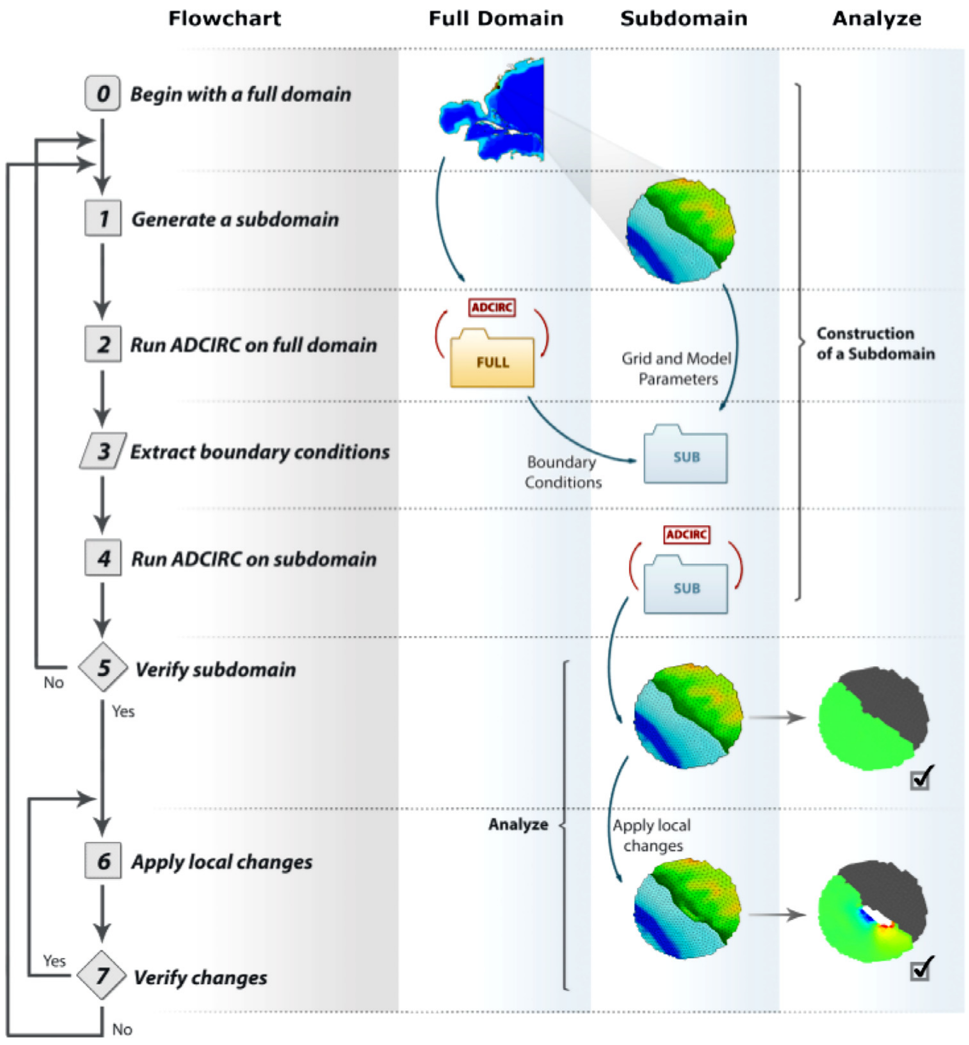


Fig. 1. Workflow for subdomain modeling [1].

inspection tools that could also be used after subdomains are modified to show that they are large enough to fully contain the altered hydrodynamics.

2.2. Design goals

Our design approach is intended to address several major goals, each aimed at improving the basic workflow and promoting the use of subdomain modeling:

1. Leverage the power of subdomain modeling with a visual interface that maintains compatibility with ADCIRC and other complementary tools that users already employ.
2. Let users work within their problem domain using direct manipulation [12], a term suggesting that interactions are more effective if the interface provides a continuous representation of the problem, the modeler manipulates this representation through physical actions (as opposed to syntax), and the interactions are both rapid and visually reversible.
3. Realize these goals through an extensible, open-source architecture designed for cross-platform expansion and future growth.

ADCIRC already has a well established community-driven software development approach, with many extensions being successfully added to the code-base, including the subdomain approach itself. However, less attention has been paid to mesh editing and visualization tools for the ADCIRC community, with a proprietary tool like SMS serving routine needs. New capabilities sometimes warrant new interface features to hook into them, so we hope to provide a base implementation for visualization and user interaction that can support the broader research and development communities and also encourage experimentation.

Our approach to extensibility is of the glass-box variety, meaning that developers who wish to extend the software can view the code, but should separate their additions from the original software in a way that does not affect the original code [13]. Such *architecture driven* frameworks typically rely on features like inheritance and dynamic binding in order to achieve extensibility, and are often easier to work with than those employing black-box extensibility.

Library dependencies and overall architecture are chosen to allow for cross-language, multi-platform growth. Not only does this promote code reuse within the ADCIRC community as researchers and developers create their own tools, it simplifies the translation of existing code onto new delivery platforms, such as web browsers and mobile devices, as the demand for tools on these platforms increases.

3. GUI design

Good interfaces allow a user to accomplish tasks as simply and efficiently as possible. In seeking to design one, we begin by observing users and their customary workflow, and look for opportunities to improve upon it.

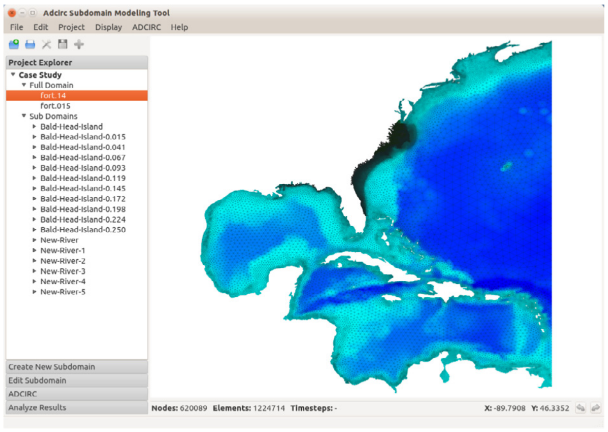


Fig. 2. An ADCIRC subdomain project in SMT.

A typical SMT user will be familiar with finite element meshes, and will often have experience using SMS as a mesh editing tool. Current users of subdomain modeling, however, are required to perform their tasks from the command line. SMT aims to improve upon the existing subdomain modeling workflow by providing a complementary interface on top of the existing subdomain modeling functionality. In order to provide a simple and intuitive interface, the layout and design of SMT draws from vector graphics editors and mesh visualization tools by grouping similar features into distinct modules.

The SMT interface is divided into two main sections, as seen in Fig. 2. On the right side of the SMT window, and occupying the majority of the available space, is the visualization pane, where a mesh is displayed and all panning, zooming, and selections are performed. On the left is a module stack containing five modules, each providing tools specific to a distinct step in the subdomain modeling process.

3.1. Visualization

The spatial refinement in ADCIRC models tends to vary, so displaying an entire mesh in a single window means that features in regions with tight nodal spacing may not be seen clearly. For example, Fig. 2 shows an SMT window with a mesh of the western North Atlantic Ocean that contains a high level of refinement along the North Carolina coast. Elements in the deep ocean, which are on the order of hundreds of kilometers in size, can be seen clearly. Coastlines, on the other hand, are much more highly refined and cannot be easily distinguished. SMT provides visual access to all parts of a large ADCIRC mesh by zooming and panning.

Subdomain modeling places no restrictions on the shape of a subdomain other than that it be a contiguous region, and SMT provides tools that allow users to design the shapes of their subdomains interactively. The following selection tools, available in the 'Create New Subdomain' module and listed by their icons in the SMT interface, give users several ways of selecting elements to be included in their subdomains:

- + - Select or deselect individual elements
- ● - Select or deselect elements by drawing a circle
- ■ - Select or deselect elements by drawing a square
- - Select or deselect elements by drawing a polygon.

Note that each of these tools includes the ability both to select and deselect elements in the full domain, allowing users to produce very complex shapes. Additionally, SMT provides undo and redo capabilities, allowing users to quickly correct inadvertent selections.

3.2. Data management

To support data management, SMT introduces the concept of a subdomain project for organizing the files associated with a full domain and any of its associated subdomains. First in the module stack is the 'Project Explorer', which shows the structure of the current subdomain project in a collapsible file tree viewer. This structure is mirrored in the underlying file system, and all files maintain their original names and formats if they have come from another tool or system (e.g., ADCIRC files). Subdomain project information is maintained by SMT in an XML file with the .spf (**s**ubdomain **p**roject **f**ile) file extension, which is used to maintain project state between SMT sessions.

3.3. Subdomain modification

The 'Edit Subdomain' module provides basic tools for editing nodal properties such as location, elevation, and Manning's n. While there is no pressing need to duplicate the more sophisticated remeshing and editing features of more commonly available tools like SMS, the SMT implementation provides robust selection tools and modular data structures that would make such extensions straightforward.

3.4. Running ADCIRC

Both full domain and subdomain ADCIRC can be run either within or outside the SMT interface. As outlined in Section 2, the second and fourth steps of constructing a subdomain model involve a full domain ADCIRC run that records values at the subdomain boundary nodes, and a subdomain run that uses the recorded data to force values along the boundaries of the subdomain, respectively. SMT ensures that steps such as these are performed in the correct order by only giving users the option to run a subdomain once the full domain run has completed and the appropriate boundary condition files have been generated by SMT.

Fig. 4 shows the options available in the 'ADCIRC' module of a project in which the full domain run has completed. The full domain is run by pressing the 'Run Full Domain' button. SMT detects when a full domain run is complete and automatically generates the boundary condition files needed to run the subdomain in a project. Any subset of the subdomains in the project can be run by selecting the desired subdomains from the list and pressing the 'Run Selected Subdomains' button. Subdomains can be edited and rerun at any time once the full domain run has completed, and duplicates of subdomains can be created and modified, allowing users to make localized changes and perform any number of individual simulations on the same subdomain.

3.5. Analyzing results

Users must be able to verify that the results of an unchanged subdomain and the full domain are equivalent. The 'Analyze Results' module of SMT provides users with the ability to compare subdomain runs with the full domain run. Differences in elevation, velocity, and wet/dry status can easily be displayed as a color gradient on the subdomain mesh, and a playback tool allows users to interactively animate these differences through every timestep of the subdomain run at which output was recorded. Additionally, users can select individual nodes in order to view the exact elevation and velocity values from both the full and subdomain runs.

After a subdomain has been modified and a simulation performed, the user must verify that variations in the hydrodynamics as a result of the modifications have not propagated to the boundaries of the subdomains, as outlined in Section 2. The same tools provided for the purpose of validating the initial, unmodified subdomain can also be used in the analysis of subdomains with modified topographies.

4. Backend development

Consistent with our intended goals, the SMT framework is designed to be flexible and extensible. The framework, which relies on object oriented design principles, is comprised of three major components. The first is the SMT backend, which is responsible for creating and maintaining subdomains and all related data. The second is the user interface framework, which is typically a platform dependent library, and the third is the graphics interface used to render ADCIRC meshes.

Fig. 5 illustrates an overall architecture that accommodates cross-language, multi-platform growth. In this paper the focus is on the desktop implementation, written in C++ and Qt [14], which uses common classes and design patterns, as described in Section 4.1, simplifying the translation to other languages and platforms. High performance visualization of ADCIRC meshes is achieved using OpenGL, as described in Section 4.2. OpenGL itself offers a cross-language, multi-platform API, so *calling code* is also easily transported across platforms. However, because user interface libraries are typically platform and language specific, the user interface component must be designed with a specific platform in mind. This structure, wherein the backend and rendering are separated from the user interface, clearly enhances flexibility and extensibility.

Connecting the backend of SMT to the user interface is straightforward using Qt's signal/slot mechanism. Additionally, the Qt object model provides convenient memory management capabilities. A number of other popular open-source user interface libraries, such as GTK+ and wxWidgets were considered, but ultimately Qt provided the right balance of user interface features and development tools, including a library specific IDE and debugger, with the added benefit of supporting multiple operating systems, allowing deployment to Windows, Mac, and Linux desktop environments.

In the following sections, we give an overview of some of the design patterns used in the backend of SMT, a description of the framework used for high performance visualization of ADCIRC meshes, and the implementation details and performance metrics of the search algorithms used for node and element selection. Additionally, we discuss possible alternative approaches and the reasoning behind the implementation decisions made during development.

4.1. Object oriented design

The structure of the SMT backend relies heavily on object oriented design principles. Their use allows us to separate SMT functionality from platform and language specific features, such as the user interface framework, as well as providing glass-box extensibility.

The underlying class structure of SMT mirrors the project structure seen in Fig. 3. The class diagram in Fig. 6 shows a small portion of the full SMT backend. We use UML [15], the Unified Modeling Language, a common notation in software development for describing the relationships between classes. In the figure we see two such types of relationships. The first is an aggregation, or a 'has a' relationship, which is indicated by a hollow diamond shape on the containing class with a single line that connects it to the contained class. The number at each end of this line, called a multiplicity, indicates the number of allowed instances of that entity in the relationship. For example, a `Project` can only have a single `FullDomain`, indicated by a 1 on the `Project` end, whereas a `FullDomain` can be a part of any number of `Projects`, indicated by a 1..∗ on the `FullDomain` end. The second is a generalization, or an 'is a' relationship, which is indicated by hollow triangle on the superclass with a line that connects it to a subclass.

At the top level of our partial class diagram is a `Project` class which is largely responsible for communicating with the user interface, as it has access to all data associated with a project. An instance of the `Project` class is required to contain an instance of a single

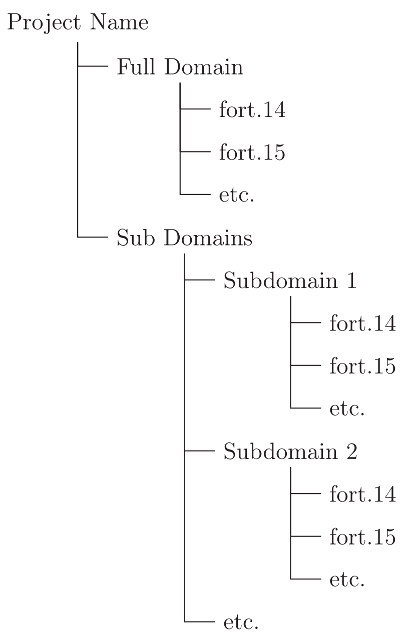


Fig. 3. The SMT project structure.

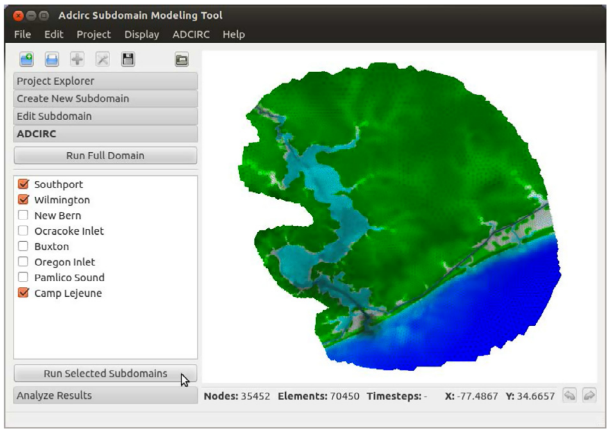


Fig. 4. A subdomain displayed in SMT.

`FullDomain` class, and may contain any number of instances of the `SubDomain` class. This directly reflects the project structure that can be seen in Fig. 3.

The `FullDomain` and `SubDomain` classes both extend the purely abstract `Domain` class. The `Domain` class provides all the general functionality needed to access the data and functionality associated with an ADCIRC run. For example, ADCIRC requires, at minimum, a `fort.14` file which contains the mesh definition and a `fort.15` files which defines model control parameters in order to perform an ADCIRC run [7]. The data associated with these files is represented in SMT by the `Fort14` and `Fort15` classes, respectively, and every `Domain` instance has access to exactly one of each. Additionally, various abstract method definitions needed by the user interface to interact with ADCIRC domains are provided by the `Domain` class. `FullDomain` and `SubDomain` inherit all of the data access functionality of their superclass, but are required to implement abstract methods like `runAdcirc()`. These implementations are unique since they both require different file configurations for the ADCIRC run to be successful. The two subclasses are also free to implement additional functionality. For example, the `SubDomain` class makes use of two additional classes, `Py140` and `Py141`, which are

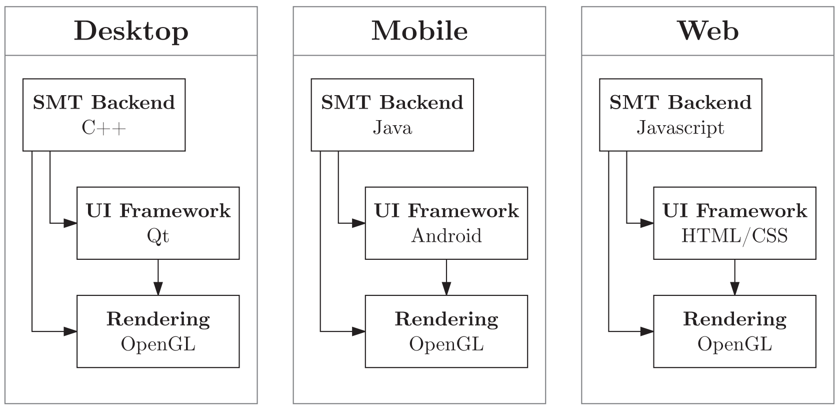


Fig. 5. An architecture that accommodates cross-language, multi-platform growth.

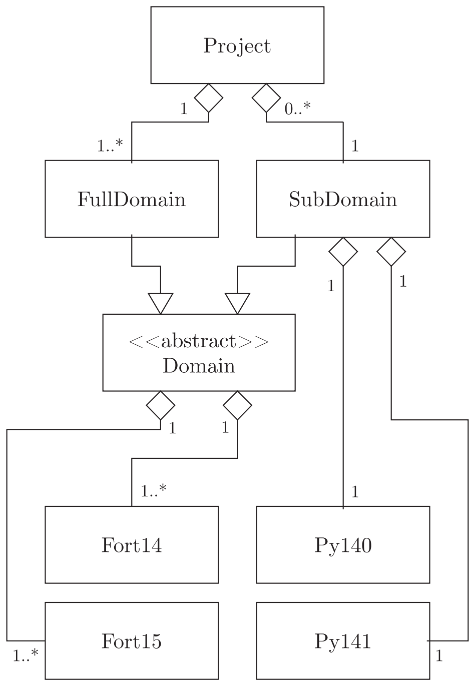


Fig. 6. SMT partial class diagram.

responsible for providing access to the `py.140` and `py.141` files, which are exclusive to subdomain ADCIRC runs.

Structuring the code using object-oriented principles provides multiple levels of glass-box extensibility. Each ADCIRC file format has a corresponding class which is responsible for I/O operations in addition to data specific functionality. For example, the `Fort14` class is capable of reading and writing `fort.14` files, as well as performing searches over mesh data. This broad level of extensibility allows developers to freely use and build upon this functionality without requiring any specific ties to subdomain modeling or SMT. Developers who require more in-depth functionality could, for example, use the `FullDomain` class and all classes that it depends on to incorporate the data and functionality of a full domain in their projects, again, without requiring any ties to subdomain modeling or SMT. Additionally, because the functionality has been implemented with standard types and common programming idioms, the backend that powers SMT is easily translated into different languages for use on different platforms.

4.2. *Visualization*

The visualization of large meshes in SMT is achieved using OpenGL, a cross-language, multi-platform API for rendering 2D and 3D graphics that is robust and well maintained. The OpenGL specification describes an abstract API, defining a number of language independent functions. In addition, the specification contains nothing on the subject of creating and managing a context for rendering, leaving this task to the underlying windowing system. Therefore, the choice of using OpenGL for rendering in SMT is consistent with the design goal of providing a cross-language and multi-platform architecture.

In OpenGL, a basic rendering primitive is the triangle, large numbers of which may be arranged as a mesh in order to represent a three-dimensional surface. In ADCIRC, the earth's surface is similarly represented using a triangulated irregular network, or TIN, which is made up of irregularly distributed three-dimensional points arranged in a network of non-overlapping triangles. This parallelism of representation further makes OpenGL a natural choice over external graphics libraries that may employ different data structures for representing surfaces.

4.3. *Search algorithm*

The choice of search algorithm used in the selection tools provided by SMT has a direct impact on user experience. The simplest approach would be to test each element one by one against the requested shape, or in other words, to perform a brute force search; this works well when the number of elements in the dataset is small, but performance becomes worse as the size of the dataset increases. ADCIRC domains, in order to sufficiently represent topography so that simulation results are accurate, are typically on the order of millions to tens of million of elements. A dataset this size can cause a brute force search to take seconds to complete on commonly available hardware. Because seamless user interaction is central to its functionality, the design of SMT strives to avoid perceptible system lags in responsiveness, and does so by employing a generalized range search algorithm to provide sufficiently fast search results.

4.3.1. *Background*

Range search algorithms are a class of algorithm used to determine the set of objects that fall inside or intersect with a query region [16]. In the context of SMT, this means the set of elements that fall within a drawn shape. The choice of range search algorithm depends on several factors, including the number of searches being performed, the shape of the query range, the dimensionality of the data, and whether the dataset is static or dynamic [17]. While one can find a variety of data structures for geometric range searches in the literature [16–18], most are specialized, having been developed to support a single type of query operation (i.e., a single shape).

In developing SMT, various libraries were considered for use as the backend for a range search algorithm. The Computational Geometry Algorithms Library (CGAL) [18], the Library of Efficient Data types and Algorithms (LEDA) [19], and the Mesh-Oriented datABase (MOAB) [20] each provide their own set of features that can be leveraged in software such as SMT. All three libraries have built in data structures designed for finite element meshes, and CGAL and LEDA both provide high performance range searches for several basic shapes. However, additional shape searches beyond those provided require that a developer learn the library and implement the search using the library's available data structures. CGAL, for example, provides the `Kd_tree` class which implements range searching using a template method and a model of the concept `FuzzyQueryItem`, which allows a range search to be performed over essentially any shape and over data with any dimensionality. This generalization is powerful, but nevertheless requires a mapping of the data being searched from its native format (e.g., an ADCIRC mesh) into one the library can understand and process. Further, while search capabilities can be crafted in existing libraries, their use adds a dependency on the library and potentially its implementation language as well, limiting the platforms that can be readily targeted.

As an alternative, we make use of a custom quadtree data structure to facilitate high performance searching in SMT. Doing so allows us to build and employ such structures without the need to modify or make unnecessary copies of the fundamental representations used by ADCIRC and OpenGL. The approach is general and versatile in that we have defined a minimum set of shape properties necessary to perform a range search over a quadtree, as outlined in Section 4.3.5. As with other tree-like structures, performance depends on balance and, thus, the spatial distribution of the dataset that it stores, yielding lookup times that vary from linear to logarithmic. In practice, however, ADCIRC meshes are typically well distributed resulting in very good performance.

4.3.2. *Data structure*

The range search algorithm operates on a quadtree structure that contains the full dataset. A quadtree is a hierarchical data structure in which each internal node (a branch) contains exactly four children [16]. External nodes (leaves) contain the actual data, which in our implementation is a set of either ADCIRC nodes or elements.

Physically, each branch and leaf in a quadtree represents a bounding box in two dimensional space. The bounding box of the root branch, or the highest level branch, is equivalent to the bounding box of the dataset that is stored in the tree. The four children of each branch in the quadtree represents a subdivision of that branch's bounding box into four equal parts.

The recursive definition of a quadtree directly translates into the recursive algorithm used to build it. Beginning with an empty leaf whose bounding box is equivalent to that of the full dataset, insert data points individually until the leaf reaches capacity. Once a leaf reaches capacity, split it into four quadrants and partition the data points accordingly, effectively turning the leaf into a branch. This process is repeated until all data points are stored in the quadtree [16]. The final structure of the quadtree will be dependent on both the size of the dataset and the maximum capacity defined for a leaf.

4.3.3. *Quadtree implementation*

The implementation of a quadtree used in SMT makes use of the composite design pattern, a way of using recursive composition such that there is no distinction between an object and its container. This design pattern is particularly useful here because, while branches and leaves both store different types of data, the way they are used to perform a search is essentially identical.

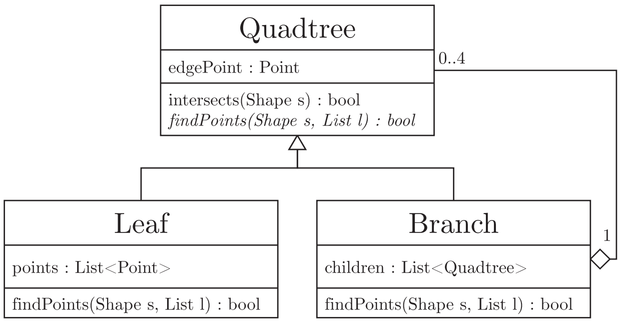

Quadtree
edgePoint : Point
intersects(Shape s) : bool
findPoints(Shape s, List l) : bool
0..4
Leaf
points : List<Point>
findPoints(Shape s, List l) : bool
Branch
children : List<Quadtree>
findPoints(Shape s, List l) : bool
1

Fig. 7. Quadtree class diagram.

The composite design pattern is implemented using an abstract class that represents both primitives and their containers [21]. Shown in the partial class diagram of Fig. 7 is an abstract `Quadtree` class that represents this container. This abstract class contains an instance variable and two methods:

- `edgePoint` - An instance variable defined as any arbitrary point that falls on the edge of the `Quadtree` instance. The SMT implementation uses the northeast corner of the bounding box.
- `intersects(Shape s)` - A method that determines if the given shape has an edge intersection with the boundaries of the `Quadtree` instance. This method uses the shape properties, as outlined in Section 4.3.5, to look for edge intersections.
- `findPoints(Shape s, List l)` - An abstract method to be defined in any concrete subclass. This method, which is passed a shape and a list, must find all data points contained by the subclass instance that fall within the shape and add them to the list. Additionally, it must return true if it can determine that the entire shape falls within its boundaries, and return false otherwise. As will be seen in Section 4.3.6, this is the method that contains the range search algorithm.

The `Leaf` and `Branch` classes are concrete subclasses of the `Quadtree` class. In addition to an implementation of the `findPoints` method, each stores a different type of data. The `Leaf` class contains a list of `Points`, which are the actual data points inserted into the quadtree. The `Branch` class contains a list of `Quadtrees`, which are the branch's children, and there will always be exactly four children in this list.

Not shown in the class diagram of Fig. 7 are the coordinates of the bounding box describing the physical location of the `Quadtree` instance. The bounding box of the root Quadtree is equivalent to the bounding box of the full dataset which it contains, and the bounding box (and thus, the northeast corner used as the `edgePoint`) of each child is calculated on the fly as they are generated through divisions along the axes. These coordinates are used in the `intersects(Shape s)` method in determining if the edge of a Quadtree intersects with the given shape.

4.3.4. *Range search theory*

A range search over a hierarchical data structure such as a tree is typically performed as follows. Starting from the root, check whether the search region intersects with (or contains) the current tree node. If it does, check the children; if not, none of the leaf nodes below this node can possibly be of interest. This quickly prunes away irrelevant portions of the dataset [17]. There are three basic properties of quadtrees that allow this type of search to be performed:

- If a quadtree falls completely within an arbitrary shape, all of the data points contained by that quadtree are also guaranteed to fall within the shape.

- If an edge of a quadtree intersects with any edge of an arbitrary shape, the data points contained by that quadtree *might* fall within the shape.
- If a quadtree does not fall completely within or intersect with an arbitrary shape, all of the data points contained by that quadtree are guaranteed to not fall within the shape.

So, in order to implement a shape search, one simply needs to be able to test for an intersection between that shape and the rectangular bounds of a quadtree. An intersection, in this case, is defined as both edge intersection and fill intersection, meaning that a quadtree fully contained inside of a shape is considered an intersection, for example.

Shapes such as rectangles, triangles, circles, or polygons, will each perform these tests in a unique way, and thus are typically given unique implementations of the range search. In order to avoid the need to implement unique search implementations for each shape, we have determined a minimum set of shape properties required by the search algorithm. Any shape which provides these properties can be used to perform a range search.

Algorithm 1: Node Search Method (Branch).

```
Method findPoints(Shape shape, List list)
1   if this.intersects(shape) then
2       for child in this.children do
3           child.findPoints(shape, list);
4       return false;
5   else
6       if this.contains(shape.edgePoint) then
7           for child in this.children do
8               if child.findPoints(shape, list) then
9                   return true;
10          return true;
11      if shape.contains(this.edgePoint) then
12          for child in this.children do
13              child.addPointsToList(list);
14      return false;
```

Algorithm 2: Node Search Method (Leaf).

```
Method findPoints(Shape shape, List list)
1   if this.intersects(shape) then
2       for point in this.points do
3           if shape.contains(point) then
4               list.add(point);
5       return false;
6   else
7       if this.contains(shape.edgePoint) then
8           for point in this.points do
9               if shape.contains(point) then
10                  list.add(point);
11          return true;
12      if shape.contains(this.edgePoint) then
13          list.add(this.points);
        return false;
```

4.3.5. Generalized shape requirements

In order to provide a range search algorithm that can be used with any arbitrary shape, a single data point and two tests are required. These three properties, which are defined in SMT through an abstract `Shape` class, are as follows:

1. The location of any single arbitrary point on the shape's edge: `Point Shape::edgePoint`
2. A test to determine if a point falls inside of the shape: `bool Shape::contains(Point p)`
3. A test to determine if a line segment intersects with the shape: `bool Shape::intersects(Point p1, Point p2)`

Note that in its implementation of the `intersects(Shape s)` method, the `Quadtree` class simply uses the third test to determine whether any of its four edges intersect with the shape.

By using an abstract Shape class, developers are fully removed from the implementation details required by the search algorithm that are not necessary for customization. Providing hooks into the search functionality through these abstractions falls in line with goal of providing glass-box extensibility, a design goal outlined in Section 2.2.

4.3.6. Performing a search

A search using the generalized range search algorithm begins by calling the `findPoints` method of the root quadtree, passing in the shape to be searched, as well as an empty list that will be populated by the data points that fall inside of the shape. If the root quadtree is in fact a branch, the code outlined in Algorithm 1 will be executed. Being a recursive algorithm, this same `findPoints` method is called on the branch's children quadtrees. If the child is another branch, the code in Algorithm 1 is executed, whereas if the child is a leaf, the code in Algorithm 2 is executed.

The tests performed in both algorithms, as outline below, are identical, but the actions performed at the result of each test differs based on whether it is being performed by a branch or a leaf.

Test 1: Edge intersection. First, a test is performed to determine whether any of the four edges of the quadtree intersects with the edge of the shape. If there is an intersection, the only conclusion that can be drawn is that any data contained by the quadtree *may* fall inside of the shape. Therefore, if the test was performed on a branch (Line 1, Algorithm 1), recursion is started on each of the branch's children. If the test was performed on a leaf (Line 1, Algorithm 2), each data point contained by that leaf is tested individually in order to determine if it falls within the shape. If it does, it is added to the list, otherwise it is ignored. In both cases, false is returned, indicating that the precise location of the shape in relation to the quadtree has not yet been determined.

If, on the other hand, it is determined that there is no edge intersection between the shape and the quadtree, one of three possible deductions can be made. Either the quadtree contains the entire shape, the quadtree falls completely inside of the shape, or the quadtree and shape are entirely disjoint. The following two tests are used to determine which of these cases is true.

Test 2: Quadtree contains shape. Because it has already been determined that there are no edge intersections, if any arbitrary point that falls on the *edge* of the shape is contained inside of the bounding box of the quadtree being tested, it can be concluded that the entirety of the shape must be fully contained inside of the quadtree. From this, it can be concluded that any data contained by the quadtree *may* fall inside of the shape.

If the test was performed on a branch, all four children are recursed upon (Lines 6–10, Algorithm 1). If the test was performed on a leaf, however, each data point contained by the leaf must be tested individually in order to determine if they fall within the shape (Lines 7–11, Algorithm 2).

Additionally, once either the recursion on all four children is complete or all data points have been tested, true is returned. This truth value indicates that the precise location of the shape in relation to the quadtree has been determined. More specifically, at this point it can be concluded that the entire shape falls completely within this quadtree, so there is no need to continue searching at any other levels of recursion.

Test 3: Shape contains quadtree. If the quadtree does not contain the shape, determining which of the two remaining possible conditions holds true can be done using a single test. Because it has been determined that there are no edge intersections, if any point on the edge of the quadtree falls inside of the shape, it can be deduced that the entire quadtree must fall entirely inside of the shape.

If the test passes and was performed on a branch (Line 11, Algorithm 1), every data point in all of the branch's children can be added to the list without performing any additional tests, and similarly, if the test was performed on a leaf (Line 12, Algorithm 2), every data point contained by the leaf can be added to the list without performing any tests. In either case, false is returned because the precise location of the shape in relation to the quadtree has not been determined.

On the other hand, if the test fails, it can be concluded that the shape and quadtree are completely disjoint and the quadtree and any data it or any of its children may contain is completely ignored for the remainder of the search.

Once the `findPoints` method call to the root quadtree returns, the search is complete and the list that was passed to the method is now populated with the subset of the data that falls inside of the shape. Additionally, the return value of the method call indicates whether or not the shape used in the search falls completely inside of the bounding box of the dataset.

4.3.7. Search performance

The design of the range search algorithm implementation used in SMT aims to provide a robust, extensible search that can provide results to the user in a near instantaneous fashion. As mentioned, studies have shown that the perceptual processing time constant, or the largest amount of time between an action and a reaction that can go unnoticed, is 100 milliseconds. Targeting this 100 ms search time provides a benchmark for testing across multiple hardware configurations and mesh sizes. Additionally, because subdomain modeling typically involves performing selections on a very small subset of a full domain, better performance on smaller selections is desirable.

The *expected* performance of the algorithm depends heavily on the structure of the quadtree being searched as well as the solution size (i.e., the number of elements found by the search). Calculating the expected performance of the search algorithm for any given scenario is difficult because it is not possible to know a priori the final structure of the full quadtree or the subset of the full quadtree that will actually be reached for any given search. However, we do know that the minimum depth of a quadtree for $N > 1$ data points is $[log_4(n)]$, indicating a perfectly balanced quadtree [22], and that a well distributed dataset, such as the nodes of an ADCIRC mesh, will produce a well balanced quadtree. Additionally, because the algorithm is a depth-first search, we know that the worst case performance is $O(|E|)$ where $|E|$ is the number of edges traversed during the search (i.e., the number of recursive calls made) [23,24]. Therefore, we can expect the performance of the algorithm to be directly proportional to the number of quadtree branches or leaves that are visited during a search. For this reason, the ordering of the tests in the algorithm is designed to avoid visiting as many unnecessary branches or leaves as possible by pruning large portions of the full dataset.

Variability in the *actual* performance of the algorithm is highly dependent on the shape being used to perform a search. The actual tests performed at each level of recursion are implementation specific, so performance of the algorithm as a whole is directly proportional to how well the `Shape::contains(Point p)` and `Shape::intersects(Point p1, Point p2)` methods perform.

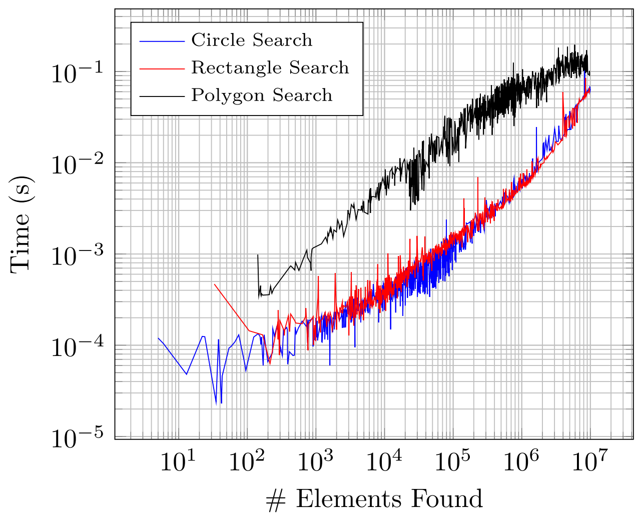


Fig. 8. Shape search performance results.

In order to demonstrate this variability, we compare the performance of our implementation of a circle, rectangle, and polygon search. The system on which all of the following tests are performed has an Intel Core i7 (2.8 GHz) CPU with 8 GB of RAM running Ubuntu 12.04. Each of the shapes is tested using an ADCIRC mesh with approximately 5 million nodes and 10 million elements, and a quadtree bucket size of 250 nodes is used. 1000 random variations of each shape are generated and the shape searches are tested using each generated shape. The polygon used in these tests are all generated with 30 edges. Each individual search is timed and the results are plotted on log-log axes, where the x-axis contains the number of elements found in any given search, and the y-axis contains the time in seconds required to find those elements.

As can be seen from the results, shown in Fig. 8, an increase in the number of elements found by a search, and thus an increase in the number of edges followed, results in a slower search, as expected. However, the circle and rectangle searches perform much better than the 30-edge polygon search. The slower performance of the polygon search can be attributed to the following two factors. First, differences in the geometric complexity of the shapes being tested can result in noticeably different levels of performance. Because circles and rectangles are very simple shapes, they are much more likely to initiate an early exit from the algorithm by falling completely inside of a branch or leaf. The 30-edge polygon, on the other hand, is much more likely to have intersections with the edges of the branches and leaves of the quadtree, making it much less likely to initiate an early exit. Second, the complexity of the `Shape::contains(Point p)` and `Shape::intersects(Point p1, Point p2)` methods has a direct impact on the performance of the algorithm as a whole. The circle and rectangle search implementations use very simple tests that require relatively little processing power. The circle search uses the Pythagorean Theorem to determine if a point falls within the circle and the parameterized equation of a line to determine if a line segment intersects with the edge of the circle. The rectangle search simply uses a bounds test to implement both methods. The polygon search, on the other hand, takes quite a bit more processing power to complete these tests. In order to determine if a polygon intersects with a line segment, each edge of the polygon is tested individually against the line segment, looking for an intersection. In the case of a 30-edge polygon being tested against a branch or leaf with four edges, this could result in 120 individual tests. In order to determine if a point falls within a polygon, this implementation uses the highly compact and very fast PNPOLY algorithm [25,26]. Performing a single test using this algorithm requires looping through every vertex of the polygon, which in this case means looping through and performing calculations at 30 individual vertices. As a result of the high complexity of these two tests as compared to the tests performed by the circle and rectangle searches, a significant decrease in performance of the algorithm is observed.

The plotted results also show that almost every search returns results faster than the 100 ms benchmark laid out in the design specifications. The exceptions are 30-edge polygon searches in which an extremely large number of elements were found. In the context of subdomains, this could be considered a fringe case, as users typically select a much smaller fraction of elements for subdomains, and the use of a 30-edge polygon to perform these selections is highly excessive. The performance curve of the algorithm is leveraged in typical subdomain modeling scenarios. Selections using simple shapes can be made in areas of extremely high refinement, which may contain millions of elements while only occupying a small percentage of the physical area of a mesh, and results appear to come instantaneously.

4.3.8. *Effect of bin size on performance*

An important factor in determining the scale of the performance of the search algorithm is the bin size used to build the quadtree. The bin size, which is the maximum number of data points added to a leaf before it is split into a branch, is directly responsible for the final structure of the quadtree. Because the performance of the search algorithm is directly related to the structure of the quadtree on which a search is being performed, the choice of bin size has an impact on the overall performance of the search algorithm, and thus the responsiveness of the selection tools.

There is an significant trade-off to consider when choosing the bin size of a quadtree. The closer the bin size is to the size of the full dataset, the less likely it is for a portion of that dataset to be pruned during the search process. Additionally, the data points contained by leaves that have edge intersections with the shape being searched must be tested individually against the shape. Therefore, a larger bin size means a larger portion of the dataset will undergo brute force testing, which is undesirable. On the other hand, the closer the bin size is to one (i.e., a leaf is only allowed to hold a single data point), the longer it takes to build the quadtree. Furthermore, an extremely small bin size will result in a very deep tree, which tends to exaggerate the quality of the `Shape::contains(Point p)` and `Shape::intersects(Point p1, Point p2)` implementations because the number of recursive calls becomes extremely high.

This performance trade-off is demonstrated in Fig. 9. Each data point, labeled by bin size, shows the average time it takes to perform a single element search using the search algorithm versus the time it takes to build the quadtree on which that particular search is performed. This data is generated from eight different quadtrees, each built using the same 5 million node, 10 million element mesh. In each of these quadtrees, 10,000 individual element searches were performed in order to calculate the average element search time.

As expected, there is a trade-off between the time it takes to build a quadtree and the average time it takes to complete a search on that quadtree. It should be emphasized that this trade-off is going to differ not only between hardware configurations, but also between meshes and shape implementations. Therefore, it is the developer who should choose a bin size that is most relevant to the application. From the user's perspective in an application such as SMT, it may be desirable to remove a few seconds of loading time at the expense of a few milliseconds of responsiveness during element selection, whereas an application that will be programmatically performing millions of searches over a single quadtree may benefit significantly from a few extra seconds of preprocessing.

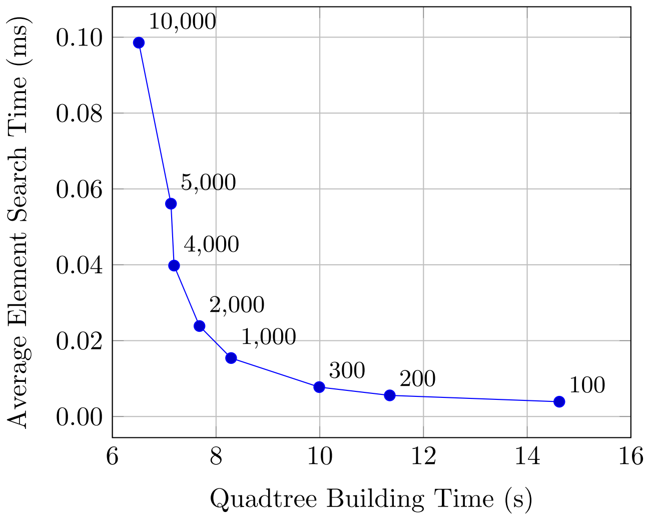


Fig. 9. Comparison of quadtree build time with average search time.

5. Case study

The following case study demonstrates the effectiveness of the subdomain modeling technique by using SMT to create and run several subdomains. In this particular example, we consider a modeler whose work is focused on two specific areas: a small portion of marsh near Bald Head Island in the Outer Banks of North Carolina, and a portion of the New River extending from its inlet on the coast of North Carolina to Jacksonville, NC.

In the case of the Bald Head Island marshes, the modeler wishes to vary Manning's n along the river bank over a number of ADCIRC simulations to determine the materials ecologically best suited to lowering water velocities during a hurricane event. The modeler expects to run at least ten different simulations, each with a different value of Manning's n in the region of interest.

In the case of the New River, she wishes to examine the effects of several possible beach nourishment projects on surge heights and water velocities through the inlet and along the river bank during a storm event. This study involves at least five different ADCIRC simulations, each with minor changes made to nodal elevations at the river's inlet.

5.1. *Creating the subdomains*

Working with subdomains in SMT begins with a new project and the selection of a full domain. A full domain is chosen by selecting a `fort.14` and `fort.15` file using the 'Create New Project' dialog, and optionally associating additional ADCIRC files with the full domain.

Once the project has been created, the modeler is ready to begin creating subdomains. Choosing an appropriate subdomain size is a matter of judgment, though subdomains of different sizes and shapes can be created and extracted simultaneously from a full domain when the extent of hydrodynamic effects of modifications are difficult to anticipate.

In the Bald Head Island region, the modeler is only varying the value of Manning's n and does not expect the resulting changes in water velocity to propagate far, allowing her to create a very small subdomain, as shown in Fig. 10a. Once the subdomain has been created, any number of duplicates can be added to the project; Fig. 10c shows eleven of them, one left unmodified for purposes of verification, and ten others named according to the new value of Manning's n that will be used in the area of interest.

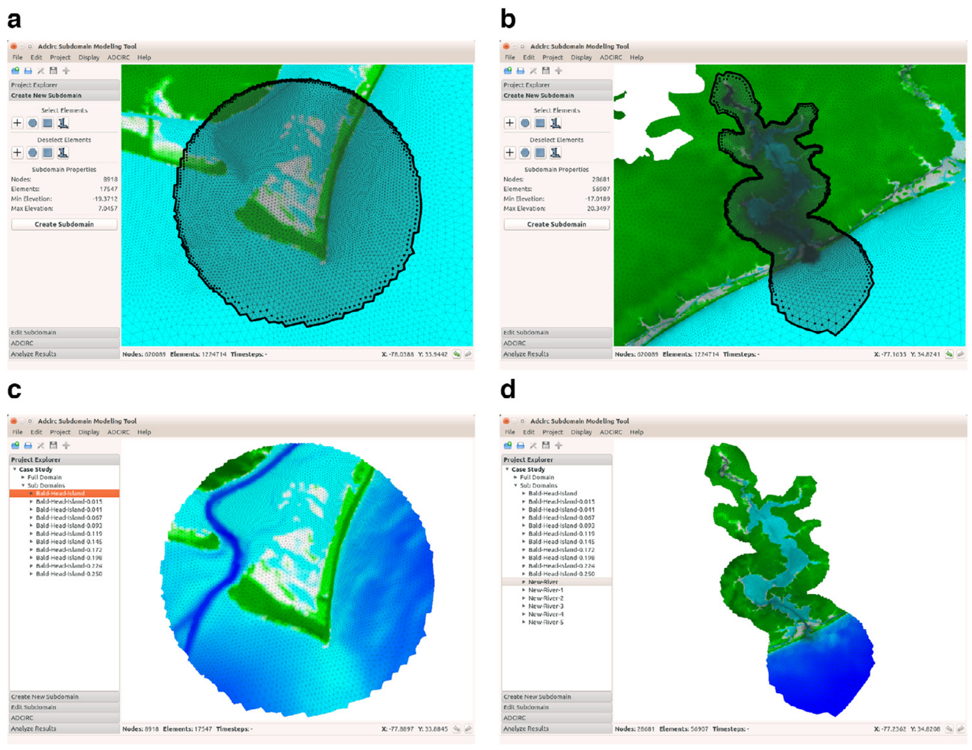
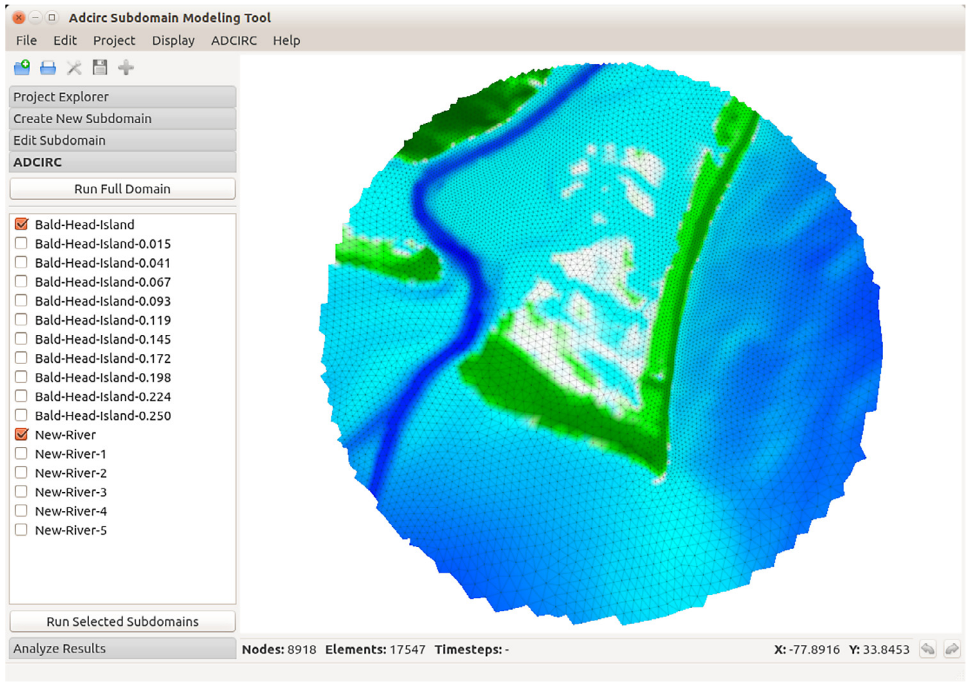


Fig. 10. (a) The modeler has selected the subdomain nodes for the Silver Lake region. (b) The modeler has selected the subdomain nodes for the Masonboro Inlet. (c) The modeler has created all eleven copies of the Silver Lake subdomain. (d) The modeler has created all subdomains needed for this study.

For the New River Inlet study area, the modeler anticipates more substantial changes will be made to the mesh resulting in a broader extent of their effects, and therefore chooses a larger subdomain size, as shown in Fig. 10b. After creating all duplicate subdomains, including one for verification, the modeler now has the full set of subdomains listed within the Project Explorer shown in Fig. 10d.

5.2. Running ADCIRC and editing subdomains

With all of the subdomains created, the modeler is now ready to perform the full domain ADCIRC run, which will record the boundary conditions for both the Bald Head Island and New River Inlet study areas. As each unique subdomain was created by the modeler, SMT updated the files necessary to initiate the recording of the boundary



- [✓] Bald-Head-Island
- [ ] Bald-Head-Island-0.015
- [ ] Bald-Head-Island-0.041
- [ ] Bald-Head-Island-0.067
- [ ] Bald-Head-Island-0.093
- [ ] Bald-Head-Island-0.119
- [ ] Bald-Head-Island-0.145
- [ ] Bald-Head-Island-0.172
- [ ] Bald-Head-Island-0.198
- [ ] Bald-Head-Island-0.224
- [ ] Bald-Head-Island-0.250
- [✓] New-River
- [ ] New-River-1
- [ ] New-River-2
- [ ] New-River-3
- [ ] New-River-4
- [ ] New-River-5

Nodes: 8918 **Elements:** 17547 **Timesteps:** - **X:** -77.8916 **Y:** 33.8453

Fig. 11. The modeler will run the unchanged subdomains for verification before running the altered subdomains.

Table 1
Comparison of the three domains used in this case study.

Domain	# Nodes	# Elements	% Full Domain	# Runs	CPU-hours/run
Full domain	620,089	1,224,714	100.0	1	1,080
New river	28,681	56,907	4.6	6	42
Bald head island	8,918	17,547	1.4	11	12

conditions during the full domain run. So, the modeler simply needs to initiate the full domain run as normal, either from the command line or using the ADCIRC module in SMT. During the full domain run, the modeler may begin making changes to the subdomain meshes using either the simple tools provided by SMT in the 'Edit Subdomain' module, or an external mesh editing tool such as SMS. It should be noted that if the unmodified subdomain is unable to be verified, these modifications will either be lost or another tool must be used to insert them into a larger iteration of the subdomain.

Once the full domain run and the verification step have been completed, any of the modified subdomain runs can be performed. SMT allows modelers to select a set of subdomain runs to be performed, as shown in Fig. 11. When running a subdomain, SMT applies the appropriate boundary conditions that were recorded during the full domain run.

A summary of the three ADCIRC domains being used in this case study is presented in Table 1. The computational benefits of using the subdomain modeling technique can be seen by comparing the full domain run to the subdomain runs and considering the number and sizes of the ADCIRC runs required by the modeler. Without using subdomains, almost 20,000 CPU-hours would be required to perform the 18 runs, in contrast with the approximately 1,128 CPU-hours actually used. The time cost to the modeler in setting up and extracting subdomains is minimal: about 15 min for each of the two study areas presented using only SMT.

6. Conclusion and future work

Simulating the effects of storm surge in hurricane-prone regions is essential for determining the weaknesses of civil infrastructure systems and identifying protective measures that can strengthen them. The science of storm surge modeling, with some refactoring, can be effectively employed for use in engineering infrastructure assessment, despite its computational demands. Using subdomain modeling, large-scale simulations can be performed on multiple, alternative topographies with much less computational cost than would otherwise be required. We describe an interface that allows modelers to take full advantage of the subdomain modeling technique through an interactive and visual environment. It removes the tedium and error-prone nature of manual subdomain creation and data management so that users are better able to focus on design and analysis of storm resilient infrastructure.

Regarding implementation, instead of creating a block-box library of functionality, the SMT codebase is written to be re-editable, providing glass-box extensibility that allows functionality to be extracted and used in essentially any language or on any platform. We have made use of this capability on multiple occasions, which speaks to the versatility accommodated by its design. We have used pieces of the SMT code to create an ADCIRC visualization and scripting program, exposing ADCIRC simulation data to a Python scripting interface, a tool that has been useful in querying ADCIRC datasets. Additionally, much of the SMT functionality has been translated into Java, with the idea that certain features could be useful in a mobile application, as well as into Javascript, as a step toward providing a full featured, web-based version of SMT.

While SMT provides a streamlined workflow, we understand that modelers may wish to rely on complementary tools to complete their tasks. For instance, in order to take advantage of the subdomain modeling workflow, one must be able to make modifications to subdomain meshes. SMT provides a very basic nodal attribute editing tool, but is not currently equipped for extensive mesh processing. Existing software, such as SMS, provide more capable mesh editing tools, and their wide acceptance within the ADCIRC community means it is likely modelers will already be familiar with their use. However, SMT has been designed with these features in mind, and two sophisticated methods of re-meshing and mesh refinement are being tested for inclusion in SMT, with the goal of inserting new physical or geometric features into subdomains. The first is a heuristic method that attempts to insert a feature into the existing subdomain mesh while preserving as much of the original mesh as possible, and the second allows a modeler to completely remove an interior portion of the subdomain mesh in order to insert a higher quality mesh feature in the empty location and generate a smooth gradation of elements in order to connect the surrounding subdomain mesh with the newly inserted feature mesh.

Other improvements left for future work include client-server accommodation that reduces potential data movement in some workflows. ADCIRC simulations are often performed in parallel on HPC clusters with extensive computing power and storage capacity, and in many cases these clusters are accessed over a network. In this case, the full domain mesh file and all subdomain files are transferred over the network in order to use them in SMT. Doing so could be limiting due to the size of the files and/or network speeds. We hope to implement a server-side SMT client that allows all data processing to remain on the server while allowing the user to interact with the data remotely through SMT.

Acknowledgments

The authors thank Rick Luettich and Brian Blanton for their helpful feedback and for providing the base files used in this study, including the NC FEMA meshes and wind files. We also thank Alper Altuntas for his work generating both data and images, and Lake Trask for his ongoing work in developing mesh editing tools for SMT.

This material is based upon work supported by the Coastal Hazards Center of Excellence, a US Department of Homeland Security Science and Technology Center of Excellence under Award Number 2008-ST-061-ND 0001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DHS.

References

[1] Baugh J, Altuntas A, Dyer T, Simon J. An exact reanalysis technique for storm surge and tides in a geographic region of interest. Coast Eng 2015;97(0):60–77. doi:10.1016/j.coastaleng.2014.12.003.
[2] Blain CA, Westerink JJ, Luettich RA. The influence of domain size on the response characteristics of a hurricane storm surge model. J Geophys Res Oceans 1994;99(C9):18467–79. doi:10.1029/94JC01348.
[3] Luettich RA, Westerink JJ. Formulation and numerical implementation of the 2D/3D ADCIRC finite element model version 44.xx. http://www.unc.edu/ims/adcirc/publications/2004/2004_Luettich.pdf, 2004 (accessed 28.10.15).
[4] Tanaka S, Bunya S, Westerink J, Dawson C, Luettich J, R A. Scalability of an unstructured grid continuous Galerkin based hurricane storm surge model. J Sci Comput 2011;46(3):329–58. doi:10.1007/s10915-010-9402-1.
[5] Westerink JJ, Luettich RA, Feyen JC, Atkinson JH, Dawson C, Roberts HJ, Powell MD, Dunion JP, Kubatko EJ, Pourtaheri H. A basin- to channel-scale unstructured grid hurricane storm surge model applied to southern Louisiana. Mon Weather Rev 2008;136(3):833–64.
[6] Luettich RA, Westerink JJ. Elemental wetting and drying in the ADCIRC hydrodynamic model: Upgrades and documentation for ADCIRC version 34.xx. Technical Report. Department of the Army, U.S. Army Corps of Engineers, Waterways Experiment Station, Vicksburg, MS; 1999.
[7] The official ADCIRC web site. http://adcirc.org (accessed 28.10.15).
[8] Altuntas A. Downscaling storm surge models for engineering applications, Raleigh, NC: North Carolina State University; 2012. Master's thesis.
[9] Aquaveo. http://aquaveo.com (accessed 28.10.15).

[10] Card SK, Robertson GG, Mackinlay JD. The information visualizer, an information workspace. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '91; 1991. p. 181–6. ISBN 0-89791-383-3. doi:10.1145/108844.108874.New York, NY, USA: ACM

[11] Card SK. The psychology of human-computer interaction. Hillsdale, N.J.: L. Erlbaum Associates; 1983.

[12] Shneiderman B. Direct manipulation: a step beyond programming languages. Computer 1983;16(8):57–69. doi:10.1109/MC.1983.1654471.

[13] Zenger M. Programming language abstractions for extensible software components, Lausanne, Switzerland: Swiss Federal Institute of Technology; 2004. Ph.D. thesis.

[14] Qt. http://qt-project.org (accessed 28.10.15).

[15] Rumbaugh J, Jacobson I, Booch G. The unified modeling language reference manual. Pearson Higher Education; 2004.

[16] de Berg M, Cheong O, van Kreveld M, Overmars M. Computational geometry: Algorithms and applications. Berlin: Springer; 2008. doi:10.1007/978-3-540-77974-2. ISBN 9783540779735 (hardcover : alk. paper); 3540779736 (hardcover : alk. paper)

[17] Skiena SS. The algorithm design manual, London: Springer; 2008. ISBN 9781848000698 (hbk.); 1848000693 (hbk.); 1848000707 (ebook); 9781848000704 (ebook). doi:10.1007/978-1-84800-070-4.

[18] Basken M. 2D range and neighbor search. In: CGAL user and reference manual. CGAL Editorial Board; 2015. http://doc.cgal.org/4.5.2/Manual/packages.html#PkgPointSet2Summary (accessed 28.10.15).

[19] LEDA. http://www.algorithmic-solutions.com/ (accessed 28.10.15).

[20] MOAB. ftp://ftp.mcs.anl.gov/pub/fathom/moab-docs/index.html.

[21] Emma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley; 1995. ISBN 9780201633610; 0201633612

[22] Samet H. Foundations of multidimensional and metric data structures. Amsterdam ; Boston: Elsevier/Morgan Kaufmann; 2006. ISBN 0123694469; 9780123694461

[23] Dyer T. An interface for subdomain modeling using a novel range search algorithm for extracting arbitrary shapes, Raleigh, NC: North Carolina State University; 2014. Master's thesis.

[24] Jungnickel D. Graphs, networks, and algorithms. Heidelberg; New York: Springer; 2013. ISBN 9783642322778; 3642322778

[25] Franklin WR. PNPOLY - point inclusion in polygon test. http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html (accessed 28.10.15).

[26] Hormann K, Agathos A. The point in polygon problem for arbitrary polygons. Comput Geom 2001;20(3):131–44. doi:10.1016/S0925-7721(01)00012-8.