RescUSim and IPython: An environment for offshore emergency preparedness planning

Markus Brachner*[1,2] and Lars Magnus Hvattum[2]

[1]Molde University College, Molde
[2]University of Tromsø, Tromsø

Abstract

Emergency preparedness is crucial for oil and gas operators. While accidents in this industry are commonly connected to oil spill disasters, helicopter accidents are, in terms of incidence rates, a more grave concern in Norway. A recent helicopter accident near Bergen has brought this subject back into focus. We introduce RescUSim, a simulator for rescue missions after offshore helicopter accidents, which is implemented as an open source library with bindings for the Python language. We discuss the modules in the existing Python ecosystem that are used for data preparation and analysis. We show how RescUSim and the interactive computing environment IPython can join forces to provide a tool for planning rescue preparedness for oil and gas related offshore activities.

1 Introduction

On 29 June 2016, around noon, a Super Puma EC225 Helicopter with two pilots and eleven passengers lost its main rotor on the way inbound from the Gullfaks B platform and crashed on the small island Turøy near the coast. In this area, which features high offshore related activity, rescue resources were on site within short time, but to no avail: no one survived the crash. Helicopter transportation is one of the major hazards for employees on offshore installations [16]. In the Barents Sea, an area under consideration for the future development of oil and gas exploration, long transport distances, sparse infrastructure, low maritime activity, and harsh environmental conditions will exacerbate the issue of transporting offshore personnel. There, relying on existing infrastructure will not be possible, and the Emergency Response System (ERS) needs to be planned from scratch in a robust but economically feasible way.

Planning and evaluating offshore emergency preparedness solutions is difficult, because of the – luckily – rare occurrence of incidents in the past. Besides running through real-life scenarios, computer simulation is one of the few possibilities to get estimates of the ERS capability [14]. Brachner [1] presented a simulation model for offshore Search-and-Rescue (SAR) operations after a helicopter ditch.

*corresponding author, Markus.Brachner@uit.no
This paper was presented at the NIK-2016 conference; see http://www.nik.no/.

This simulation is – unlike Human-in-the-loop simulations like the one presented by Xiuwen [17] or VSTEP's RescueSim (`http://vstepsimulation.com/product/rescuesim/`, accessed Oct 18 2016) – not targeted at training SAR personnel, but at evaluating the capabilities of an ERS, that is, to check if the right amount and types of rescue units (RUs) are placed at the right locations to provide enough rescue capacity in case of an incident. We define rescue capacity as the number of persons that can be rescued within a given time at a given incident location. This capacity can be obtained collaboratively by several RUs. That is, if several RUs are within range, all of them can contribute to the rescue operation. In the simulation the available RUs are mobilized, travel to the incident location, and conduct a rescue operation by picking up persons from the sea. Environmental conditions and other stochastic factors influence mobilization, travel, and pickup time. Many of the assumptions made when developing the simulation model are based on the report that was used as a basis for the Norwegian Oil and Gas guidelines for establishing area preparedness [15] and on work done by SINTEF [7].

The simulation based on this model, which was implemented in the Python programming language, provided the desired results, and the run time of up to several minutes was still adequate for single evaluations of ERS configurations [1]. While Python allowed us to develop the model quickly, it became clear as new demands and requirements were emerging that the existing code base with no decent software architecture in mind could only serve as a close-ended prototype. In addition to the SAR helicopters originally modelled, we wanted also to include maritime RUs, that is, Emergency Rescue Vessels (ERVs). The simulator should be easily extendable with new RU types. The architecture should also allow for using the same data model with different simulator engines, as we wanted to experiment with alternative algorithms and a heterogeneous computing engine utilizing the Graphical Processing Unit. Furthermore, we wanted to utilize a good deal more weather data instead of the one year span used in the original model, to account for annual variations and climatic trends. Finally, the simulator run time should be drastically cut in order to allow for the evaluation of a big amount of scenarios within a short period of time in order to utilize the simulator in a simulation-optimization framework to support optimization of the ERS design.

For these reasons, the simulator was re-implemented from grounds up. Still, the Python ecosystem provides a rich set of libraries that are particularly useful for ERS planning. Very elegant solutions for working with spatial data, processing input data, and analyzing the output data are available in this environment. This made us transfer the computational intensive simulator core to an external C++ library with Python bindings, such that the best of both worlds could be utilized. While the main focus in [1] was the model itself, in this paper we present the re-implemented and extended library together with third-party components that form our environment for ERS planning and evaluation.

The remainder of this paper is organized as follows: In Section 2 we describe the work flow that we intend for interacting with the simulator and derive the functionality that is required from the environment. In Section 3 we map this functionality to concrete third-party packages in the Python ecosystem where available. We give an overview over these packages and describe the RescUSimCpp library, which covers the simulation functionality that have been added. Following this, in Section 4 we give a concrete example of the usage and show how to set up

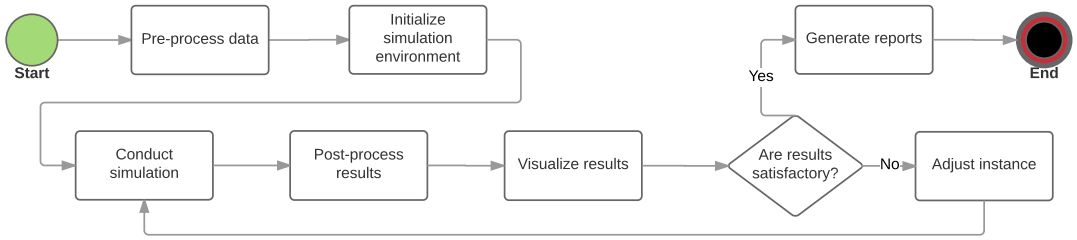


Figure 1: The intended work flow for offshore preparedness planning.

and run a simulation, and we compare the run-times of the new implementation to the one we used earlier. We conclude the paper with Section 5, where we discuss alternatives to the presented environment and give some directions for future improvements.

2 Work flow and functional requirements

Figure 1 shows the intended work flow for using the simulator. The aim was to provide an environment that facilitates this process, but still is flexible enough to give the user the freedom to deviate from it. Below we describe the separate tasks and list the functions that are required to execute them.

The simulation uses historic weather data from the Norwegian meteorological institute [11]. In the pre-processing stage this weather data is prepared for further use. One feature of emergency preparedness is the operation in a wide area, where earth curvature needs to be taken into account. For the sake of efficient computation, we therefore work with projections into the Euclidean plane. Data sources, particularly the weather data, may come in one particular projection, but need to be re-projected into another one in order to work on one single projection; this operation is called *warping*. Thus, functions for conducting map projections and warping are needed.

To be able to handle large amounts of weather data, we need to compress the data by reducing the number of data points and cropping the data to the region under consideration. This requires the use of efficient in-memory as well as storage data structures combined with tools that allow efficient spatial queries for interpolating the values for the target grid nodes.

In this stage it is also possible to add data that are not collected, but pre-computed. In our model we use the daylight conditions as an input for the rescue operation. Those can be obtained by astronomical calculations; they involve computationally intensive operations, but the results can be used repeatedly. It is therefore better to put these in the pre-processing stage, rather than to include the computations in the simulation.

Finally, the pre-processed data need to be represented in a format that is easily store- and retrievable. Array slicing from permanent storage should be supported in order to retrieve spatial and temporal subsets of the weather data without the need to load the full data-set into working memory.

The stage of initialization involves instantiating the simulator, loading weather data, and providing the set of points where the simulation will measure the rescue capacity. Furthermore, at this stage the user decides about the amount, type, specification and location of RUs by instantiating them and setting their

properties. Measuring points could be located along helicopter transport paths or within a defined area under consideration. As an increased number of measuring points also increases simulation time, spatial analysis including set-theoretic and constructive methods for calculating envelopes and convex hulls of given points, creating geometric figures and discretizing them, and performing operations on geometric sets are used for reducing the amount of measuring points to the minimum.

After the simulator is set up, the actual simulation is conducted, that is, the rescue operations to the specified incident locations are simulated, returning the number of people that could be rescued within a given time limit.

The task of post-processing involves the transformation of the raw simulator results into the target structure that should answer the questions to the simulation. This may require in-memory array slicing, aggregation operations over time or space, such as calculating the average or worst case rescue capacity at the measurement points, or filtering results to identify the time or location at which a minimum rescue capacity was undercut.

Furthermore, these results may be visualized. This not only requires basic functions like bar-charts, scatter-plots, or histograms, but also cartographic visualization, which is common to emergency management activities [12]. These maps could include heat-map overlays, geometric shapes, and annotations.

The simulation can be run with adjusted parameters until the results are satisfactory. A requirement may be, for example, that an area or transport path under consideration is sufficiently covered by RUs, that is, enough rescue capacity is provided for any point within this area or path. Since we see the work-flow as an iterative process, it is of advantage to reuse results from former runs of the simulation such that only components that are adjusted need to be recomputed. In this way the computational time can be further reduced.

As a final step, the execution process and the obtained results should be documented such that they are reproducible. The environment should provide functionality to do this in an easy way. Moreover, results – particularly graphical visualizations – should be exportable for creating reports.

3 Python packages

In this section we discuss the central Python modules that are used to support the described work flow. We use *Anaconda* (`https://continuum.io`, accessed 18 Aug 2016), a Python distribution that is specifically intended for scientific computing. It already contains a big part of the needed packages and makes it easy to add the ones which do not come as a part of the distribution. All of the described packages are licensed under open source – most of them BSD-like – licenses, which allows use and redistribution with minimal requirements.

RescUSimCpp library

The RescUSimCpp library is the simulation core that was programmed in C++ to reduce the required computational time. It provides bindings to Python using pybind11 (`https://github.com/pybind/pybind11`, accessed 17 Aug 2016), a library that exposes C++ types in Python and vice versa. The source of the library is available at `https://github.com/mbrachner/RescUSim`.

This library simulates rescue missions at given incident locations and returns the rescue capacity. For each RU, the mobilization, transit, and pickup phases are

simulated and the resulting individual rescue capacity is consolidated by accounting for possible interference effects between RUs. The mobilization time can be specified by a constant value or a probability distribution. The transit time to the incident location is calculated by iterative dead-reckoning, that is, by advancing the RU stepwise from origin to destination, summing up the individual times that a RU needs for completing the steps considering the local weather conditions. For the transit of SAR helicopters the wind conditions are taken into account in the way described by Brachner [1]. When simulating the transit of ERVs the wave height is influencing the travel time [5]. The pickup of persons from the sea is modeled after estimates by Kråkenes et al. [7].

Figure 2 shows the basic architecture of the simulator library. The *Simulator* class is the central part and is the superclass of specific simulator implementations. At the moment, two simulators are implemented. The first one is *SimulatorCPU*, which simulates the ERS using the CPU. As many demand points have to be evaluated in one run, the simulation allows a high degree of parallelism which is exploited by using the OpenMP API (`http://openmp.org/`, accessed 16 Aug 2016). In this way it is possible to use all available cores on the CPU. The other implementation *SimulatorOpenCL* uses the GPU to conduct the simulation. A Simulator is associated to an instance of a Weather class, which is used to load the weather data from persistent storage into memory and retrieve the weather data for a given point.

Each RescueUnit entity is specified by a data holding object that represents its performance parameters. Its behavior, that is, which computations to perform on its performance parameters, are separated. As a consequence, for the CPU implementation the *ERV* and *Helicopter* data objects are associated to *ERVCPU* and *HelicopterCPU* objects. For the OpenCL implementation the behavior is part of the SimulatorOpenCL class. This is mainly because OpenCL provides a programming language that is based on a standard C language specification (C99) and thus does not provide object orientation. One more reason is the interaction between host and GPU, which makes very different programming approaches necessary. For the user, who only interacts with data objects, the separation of data and behavior is transparent. As soon as a data object is added to a simulator instance, it is handled in the simulator-specific way, that is, it is either wrapped by the appropriate behavioral classes or directly linked to the simulator instance.

The separation of data and behavior adds one more advantage: The data holding objects can be also used as entities for optimization problems like the one presented by Brachner [2] or Razi [10], linking optimization and simulation of ERS.

Third party provided packages and projects

IPython together with *Jupyter* (`https://ipython.org/`, accessed 25 Aug 2016) are the fundamental components for interacting with RescUSim. IPython is built upon Python, adding interactive features such as tab-completion, object introspection, and a help system. A central feature is the Jupyter Notebook which very recently was spinned off as a separate project to support additional languages. This is a web application that allows to create and share notebook-like documents that unify executable code snippets and documentation by inline visualizations, explanatory text, and equations. While this provides the interactivity and flexibility needed to work with the simulator, it streamlines the process of documenting the work and

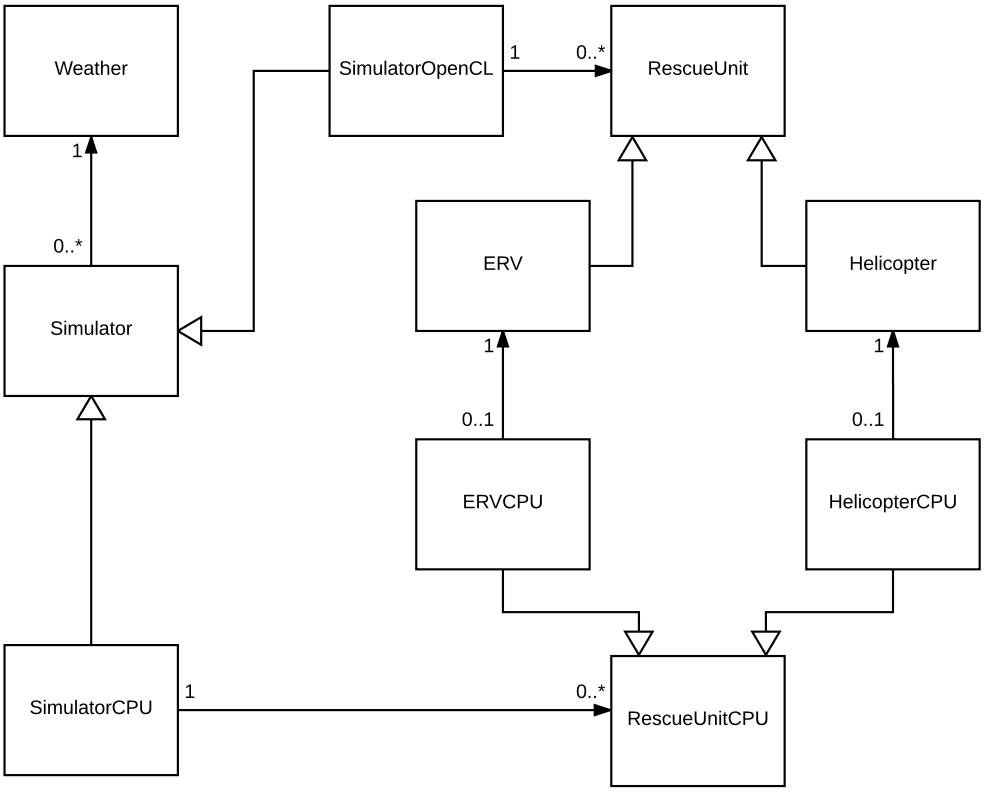


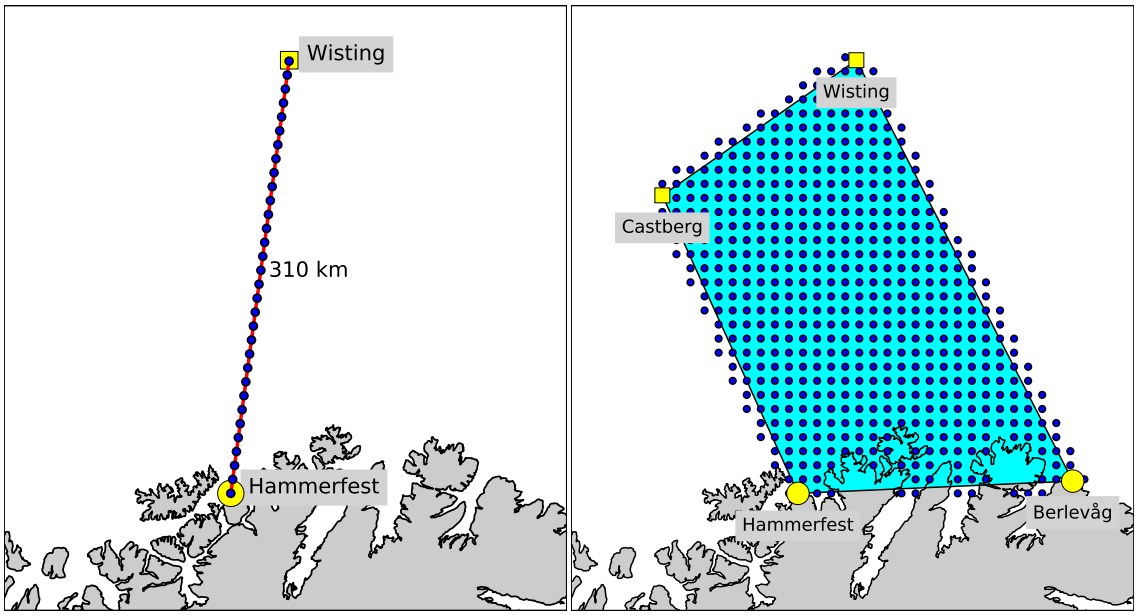
Figure 2: UML class diagram showing the core classes of the simulator.

presenting the results practically alongside the process to ensure reproducibility.

NumPy [13] provides a data structure for efficient manipulation and processing of multi-dimensional arrays. We use it for working with weather data, and the simulation returns its results as a NumPy array, which makes it easy to aggregate and filter the results during post-processing. This in-memory data structure plays very well together with *HDF5* and a package that provides Python bindings via *h5py* (http://www.h5py.org/, accessed 25 Aug 2016), which allow to store, manipulate, and retrieve a big amount of array-structured data.

The simulation results are analyzed and visualized using *Matplotlib* [4]. This package includes also the *Basemap* toolkit for plotting data on maps, along with the *proj.4* library to project geographic coordinates into the plane. In addition, the functionality of this projection library is exposed such that arbitrary geographic data can be projected without visualizing it. This streamlines working with geographic data: once a map is created, converting geographic to planar data and data visualization happens through this object using the same projection.

Once the geographic coordinates have been projected onto the plane, *Shapely* (`http://toblerity.org/shapely/`, accessed 25 Aug 2016) facilitates manipulation and analysis of geometric objects. This package defines points, curves, and surfaces as fundamental geometric types and allows to conduct a wide range of set-theoretic, construction, transformation, and merging operations on these objects. As an example, the measurement points for the simulation as shown in Figure 3 can be constructed intuitively and with only few lines of code.



(a) Discretization of a transport path.

(b) Discretization of an area defined by the convex hull of a set of given facilities including a buffer, excluding points on the landmass.

Figure 3: The Shapely library facilitates the discretization of measurement points. Even complex geometrical operations are conducted by few lines of code.

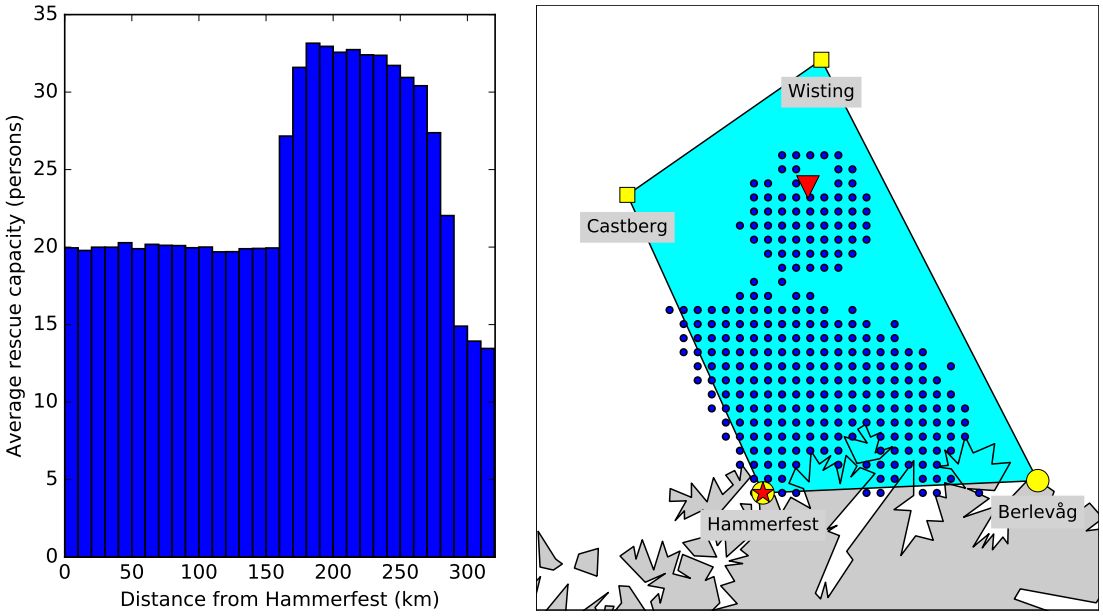
4 Usage

Listing 1 shows the full process of setting up and running a simulation. Figure 4a presents the exemplary analysis of the simulation results that could assist in planning decisions. For instance, rescue capacity could be better distributed along a transport path, given that the number of persons to rescue is not varying along the route. Alternatively, analyzing a whole area as shown in Figure 4b would allow for adapting the transport paths to avoid regions that do not provide sufficient coverage, similar to what is described by Brachner [2] in a deterministic variant. These two figures highlight also, that arbitrary criteria – in the presented case average rescue capacity, and required fraction of successful rescue missions – can be applied to decide which incident locations can be considered to be sufficiently covered. By decoupling instance preparation, simulation, and analysis and conducting the first and the last step in IPython the presented environment provides the freedom to adapt the process to the specific questions that need to be answered.

Table 1 shows the run time for the instances as specified by Brachner [1]. The results of the former implementation have been obtained by re-running these instances with the simulation model described in the referred paper with more recent hardware, which is a dual-core Intel Core i5-4210U CPU with 8GB of working memory. The results show that we achieved a speed-up of approximately factor three. As the new library is also able to utilize multiple processor cores, we report the run-times of the library with OpenMP disabled in order to show the effect of parallel computing. The computation scales well. Moreover, as the comparison between running the simulation with two and four threads shows, the simulation can also take advantage of the hyper-threading feature of the dual-core processor.

Listing 1: Full example for initializing and running the simulation.

```
# Imports
import numpy as np
import RescUSimCpp
from mpl_toolkits.basemap import Basemap
from shapely.geometry import LineString, MultiPoint, Point
from shapely.ops import cascaded_union

# Initialize a dictionary with off- and onshore facilities
poi = {name :         {'coords':c,            'type':s} for name,c,s in [
      ['Hammerfest',  (23.768302, 70.701319), 'helibase' ],
      ['Berlevaag',   (29.090389, 70.854502), 'helibase' ],
      ['Wisting',     (24.232358, 73.491134), 'platform' ],
      ['Castberg',    (20.347568, 72.494341), 'platform' ],
]}

# Initialize map. This map is not only used for visualization, but also
    for projecting geographic coordinates into the euclidean plane.
bmap = Basemap(projection='aeqd', lat_0=72, lon_0=29, resolution='l',
                  llcrnrlon=15, llcrnrlat=69,
                  urcrnrlon=41, urcrnrlat=75.6, area_thresh=100)

# Create a transport path from Hammerfest to the Wisting facility
path = LineString([   bmap(*poi['Hammerfest']['coords']),
                      bmap(*poi['Wisting']['coords'])])

# Discretize path into measuring points 20 km apart from each other
measure_points = MultiPoint([path.interpolate(d)
   for d in np.arange(0,path.length,20000)])

weather = RescUSimCpp.Weather("c:\\tmp\\data_idw.h5") # Load weather
sim = RescUSimCpp.SimulatorCPU(weather) #Initialize simulator

sim.addStationaryRU( # Add stationary rescue units:
   RescUSimCpp.Helicopter("Heli1",weather) # One SAR Helicopter
      .setPos(*bmap(*poi['Hammerfest']['coords']))) # ... at Hammerfest,
sim.addStationaryRU(RescUSimCpp.ERV("ERV",weather) # ... and one ERV
   .setPos(393174,350706)) # ... inbetween Hammerfest and Wisting.

# Create for each point a sample of 1000 incidents at different times
incidentArray=np.array([(p.x,p.y,r) for p in measure_points_path
  for r in np.random.random_sample(1000)*(weather.getNumScenarios()-2)])
sim.addIncidents(incidentArray); # Pass the incidents to the simulator

# Conduct simulation
res=sim.simulateResponseSample()
```



(a) Average rescue capacity along a transport path

(b) Incident locations, where 95% of the rescue missions were successful

Figure 4: Results of the simulation experiments

Table 1: Run times for executing the simulation on the instances specified by Brachner [1] in seconds. The reported values show the best out of three runs, respectively.

Scenario	Former implementation	RescUSim full run without OpenMP	RescUSim full run with OpenMP 2 threads	RescUSim full run with OpenMP 4 threads	RescUSim recalculation with OpenMP 4 threads
(a)	39	27	16	12	7
(b)	107	81	48	35	8
(c)	104	88	50	37	8
(d)	110	75	46	33	8

We provide also the results of re-running the simulation after re-locating a rescue unit to a new position: The new simulation library features the caching of intermediate results. Consequently, simulations that are re-run will only simulate the response of RUs that changed in any of its parameters, for example its position, or its maximum speed. Thus, computational time is considerably reduced when searching for the optimum of a single decision variable, for instance, the best position of a RU, assuming the other RU positions to be fixed.

Some comments have to be made when comparing the new library with the former solution. In the earlier implementation we approximated the travel times by simulating several radial travels starting from the RU origin and then interpolating the resulting travel times to the measuring points. This results in less accurate travel times the further the measurement points are located from the origin, because of the increasing distance between two radials. In RescUSimCpp the travel to each single measurement point is simulated, which increases accuracy. Furthermore, in the former implementation, the weather data was taken from the nearest point on a 10x10 km grid. However, in the new simulation the 50-fold amount of weather data does not allow for such a fine-grained grid. Thus, the grid spacing is increased to 20x20 km, but the weather data on a certain point is now determined by bi-linear interpolation from the four surrounding grid nodes.

We also performed tests using a GPU-implementation on an NVIDIA GeForce 840M GPU. This implementation was straight-forward and not particularly optimized for GPU, with one thread per measurement point. The run time turned out to be sometimes on par, but mostly worse than for the CPU implementation. However, simulating a series of incidents that have the same incident time, but vary by location, reduced run times by more than 80%, if the locations were close to each other. In our opinion, this points to a bottleneck at the GPU global memory: the reason for the speed-up with constant incident time seems to be the fairly large (1 MB) L2 memory cache of the GPU, where data locality can be exploited with incident locations that are near to each other.

5 Concluding remarks

In this paper we presented an environment, which we believe to be a useful tool to support planning and evaluation of ERSs for offshore helicopter ditching. However, there exists other options to achieve the desired results.

For the similar purpose of evaluating an ERS, Jakobsen [6] used MATLAB and Simulink with the SimEvents discrete event simulator. These products are tightly integrated with each other, as they are developed by the same software provider. Though we have not done a full implementation, we are convinced that the required functionality for data processing and visualization is fully available in MATLAB. However, some prefer Python in terms of clarity and functionality [3]. Moreover, using a discrete event simulator like SimEvents for the rescue operation involves unnecessary overhead due to the relatively simple process with little interaction between the simulated entities. Last, but not least MATLAB, Simulink, and SimEvent are commercial closed-source products that cause considerable cost even for academic licenses, while Python is free and open-source.

ArcGIS is a Geographic Information System, that would cover well the aspects of spatial processing and map visualization. It is also a commercial closed-source product. The use of scripting languages, out of which Python is the most popular,

is well supported. However, we would have needed too little of the functionality of this feature-rich solution to justify the cost. Nevertheless, our environment does not exclude ArcGIS, which can rather be seen as an add-on option.

Finally, an alternative would have been to implement the full functionality in C++. There are several reasons that speak against this approach. While compiled C++ code runs considerably faster than interpreted code, this generally involves longer development times and requires more verbose code compared to scripting languages [9]. Moreover, it distracts the user from the actual problem-solving process, with an increased need to deal with implementation issues [3]. Finally, in a pure C++ implementation we would have missed the possibility of interaction such as adjusting parameters, repositioning or adding RUs, or analyzing various aspects of the results without recompiling and rerunning. While we did not want a full-blown graphical user interface, it was important to target an easy and agile way of interaction. Any user interface in a C++ only solution would have needed to be built additionally to the simulation core. In contrast, with IPython and Jupiter the Python ecosystem provides interactivity for free, so to speak.

The RescUSim environment will be developed further. We intend to combine the simulation with meta-heuristic optimization in order to find robust ERS configurations that can cover flight paths under varying environmental conditions. We also see a potential to extend the simulator to other domains, for instance land based missions. This would require to include new types of RUs. Furthermore, a more advanced visualization of the results, for example an interactive, 3-dimensional visualization using CesiumJS (`https://cesiumjs.org/`, accessed Oct 18 2016) would improve human-computer interaction. Finally, there has been some research to model the reliability of SAR operations in more detail [8], which may be taken into consideration in our simulator.

Acknowledgements

We would like to thank three anonymous reviewers for their effort in helping us to improve the manuscript.

References

[1] M. Brachner. A simulation model to evaluate an emergency response system for offshore helicopter ditches. In *Proceedings of the 2015 Winter Simulation Conference*, pages 2366–2377. IEEE Press, 2015.

[2] M. Brachner and L. M. Hvattum. Combined emergency preparedness and operations for safe personnel transport to offshore locations. *Omega*. Accepted 19 March 2016, `http://dx.doi.org/10.1016/j.omega.2016.03.006`.

[3] H. Fangohr. *A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering*, pages 1210–1217. Springer Berlin Heidelberg, 2004.

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[5] S. R. Jacobsen and O. T. Gudmestad. Long-range rescue capability for operations in the barents sea. In *ASME 2013 32nd International Conference on*

Ocean, Offshore and Arctic Engineering, pages V006T07A013–V006T07A013. American Society of Mechanical Engineers, 2013.

[6] M. F. Jakobsen. Assessment of helicopter emergency response capacity in the barents sea. Master's thesis, Norwegian University of Science and Technology, 2015.

[7] T. Kråkenes, S. Håbrekke, I. Wærø, and K. Øien. Estimated rescue times for persons from sea. Technical report, SINTEF, 2013. Internal memo.

[8] L. Norrington, J. Quigley, A. Russell, and R. Van der Meer. Modelling the reliability of search and rescue operations with bayesian belief networks. *Reliability Engineering & System Safety*, 93(7):940–949, 2008.

[9] L. Prechelt. Are scripting languages any good? A validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. volume 57 of *Advances in Computers*, pages 205 – 270. Elsevier, 2003.

[10] N. Razi, M. Karatas, and M. M. Gunal. A combined optimization and simulation based methodology for locating search and rescue helicopters. In *Proceedings of the 49th Annual Simulation Symposium*, ANSS '16, pages 5:1–5:8, San Diego, CA, USA, 2016. Society for Computer Simulation International.

[11] M. Reistad, Ø. Breivik, H. Haakenstad, O. J. Aarnes, B. R. Furevik, and J.-R. Bidlot. A high-resolution hindcast of wind and waves for the north sea, the norwegian sea, and the barents sea. *Journal of Geophysical Research: Oceans*, 116(C5), 2011. C05019.

[12] W. A. Schafer, J. M. Carroll, S. R. Haynes, and S. Abrams. Emergency management planning as collaborative community work. *Journal of Homeland Security and Emergency Management*, 5(1), 2008.

[13] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[14] J. E. Vinnem. Evaluation of offshore emergency preparedness in view of rare accidents. *Safety Science*, 49(2):178–191, 2011.

[15] J. E. Vinnem. Retningslinjer for områdeberedskap - underlagsrapport, forutsetninger og faglige vurderinger. Accessed Oct 18, 2016. `https://www.norskoljeoggass.no/Global/Retningslinjer/HMS/Beredskap/064%20-%20Underlagsrapport.pdf`, 2012. In Norwegian.

[16] J. E. Vinnem, T. Aven, T. Husebø, J. Seljelid, and O. J. Tveit. Major hazard risk indicators for monitoring of trends in the norwegian offshore petroleum sector. *Reliability Engineering & System Safety*, 91(7):778 – 791, 2006.

[17] L. Xiuwen, X. Fangbing, and J. Yicheng. A prototype of marine search and rescue simulator. In *Information Technology and Computer Science, 2009. ITCS 2009. International Conference on*, volume 1, pages 343–346, July 2009.