# PyDES: A Framework for Complex Scheduling Analysis using Discrete-Event Simulation

**Roger A. Stuckey**
*Defence Science and Technology Organisation, Eveleigh, NSW 2015, Australia*
Roger.Stuckey@dsto.defence.gov.au

**Abstract.** This paper presents a software framework for complex scheduling analysis with a primary focus on the domains of capability development and acquisition support. The framework provides a system within which problem-dependent application models with stochastic components and processes can be constructed. The framework and models are written in the Python programming language, making the development of new models relatively simple whilst allowing for complex interactions to be implemented. For any application, discrete-event simulations of the model can be run many times in a Monte Carlo experiment to derive statistical measures of performance and effectiveness. It is also possible to conduct sensitivity analyses and explore the parameter space of application scenarios by running different sets or batches of simulations. The software comprises a core library for running each simulation, as well as libraries for the management of multiple simulations, graphical output generation and the creation of user interfaces. It can be driven with scripts from the command-line environment, or controlled via a front-end graphical user interface. The framework can also exploit multi-core systems and run in a distributed client-server architecture either locally or remotely on the Amazon Elastic Compute Cloud (EC2) for scalable performance gains. Two exemplary applications are discussed: Naval Area Defence (NAD) and; Unmanned Maritime Operations (UMO). The NAD application has been developed to support decisions regarding system integration and operations for surface-air defence of naval assets. The UMO application was developed to assess various concepts of operation, efficiency and crew workload in the use of unmanned maritime systems for large-area survey missions. The general features of the software are presented, along with some examples of its use in each of the applications.

## 1. INTRODUCTION

PyDES is a software framework for complex scheduling analysis using discrete-event simulation (DES). PyDES provides capabilities for sensitivity analysis and exploration of the parameter space, as well as a graphical user-interface and facilities for visualisation of the results. At the heart of the framework is a simulation engine capable of coordinating the response of any number of resource-based models tasked with a list of objectives to meet. In general terms, the simulator makes repeated attempts to schedule engagements against each of the outstanding objectives in a time-optimal manner, until all objectives have been engaged and all other requirements have been satisfied.

PyDES is written in the Python [1] programming language and designed for ease of implementation and extensibility. Python was chosen as the core language because it is highly abstracted and expressive, making model development relatively simple, yet powerful enough to run thousands of Monte Carlo simulations [2] with internal optimisations involving hundreds of models. It is platform independent and there is also a large community of Python programmers around the world - particularly in the scientific domain - enabling the leverage of many open-source projects, such as Numpy [3, 4] for numerical computation, Matplotlib [5] for plotting and Pyro [6] for distributed computing.

With most visual programming software solutions, often only the most basic types of processes and interactions, such as generic servers and queues, can be implemented using the graphical user-interface. Constructing a realistic simulation from these graphical building blocks can be a challenging task and the resulting models an elaborate hierarchy of blocks within blocks. Moreover, the developer will invariably reach a point at which scripts must be written within each model in order to codify more detailed behaviours. Sometimes, the integration of complex interactions can be extremely difficult or even impossible. With PyDES, the models 'live' within the code and are not limited by any graphical representation. The developer has complete control over the models, their interactions and even the procedures for scheduling and engagement in the event queue. The simulation and models also benefit from the object-oriented nature of the language with features such as encapsulation, polymorphism and inheritance.

Rather than offering a 'one-solution-fits-all' approach to modelling in the discrete-event time domain, PyDES is aimed at addressing a particular class of problem. That is, as noted above, the repeated scheduling of resource-based engagements against a list of objectives in a time-optimal manner. This paper will first present a description of the scheduling approach used and the representation of the scenario specific to each application, followed by a more detailed description of the main classes used by the simulation. Last, the issue of computational speed is discussed, and the scalability features of the framework designed to mitigate those.

## 2. DESCRIPTION OF THE PROCEDURE

In generic terms, a scenario is composed of a number of hosts - each of which holds a hierarchy of resources - and a number of generators - each holding a list of objectives. The term objective is loosely defined and dependent on the application being modelled. Every objective, once activated, must be engaged by a set of

resources, typically resident on one or more of the hosts. This is accomplished by scheduling engagements, which encapsulate a combination of start-times and durations required by each resource in meeting the objective. These engagements can be managed individually, by each host, or in a coordinated manner across several hosts. In PyDES, each Engagement Manager can be associated with several hosts, but each host can only have one Engagement Manager.

Nominally, the resources must be scheduled according to an engagement profile. The form of that profile being based on the invoking event. Examples of the engagement profile for both NAD and UMO applications are presented in the following section. There are potentially many ways by which a set of resources can be selected to meet an objective. The most naive approach will just select the first, or randomly, from each type of resource available at that time. However, there is no guarantee that all of the resources will be available at the appropriate time. Another approach is to select a set and then wait until they are all available, but this can lead to large delays or an inability to engage every objective. A better approach is to search through all combinations of resources to determine one that will engage the objective in a minimum time. This exhaustive approach will yield an optimal set, but may take some time to process, particularly if there are a substantial number of resources. Therefore, the best approach will attempt some form of combinatorial optimisation for each engagement. One of the main features that differentiate PyDES from many existing DES packages is its ability to perform this inner-loop optimisation of the resources selected. That can range from an exhaustive search for the best possible combination of resources, or a search of the subset defined by some heuristic. In PyDES, the resources are first sorted according to their idle-time and then some number of the top combinations, defined by the user, chosen for exhaustive search.

Another level of complexity arises if there are several objectives to be engaged at any one time, or another event has prompted a reassessment of the currently scheduled engagements. The objectives must be prioritised, and a non-conflicting set of resources scheduled against each. There are currently two approaches employed by the Engagement Manager:

1. Schedule an engagement for the highest priority objective if and only if resources are available at the time of assessment, plus a check at some future time when resources are expected to be available next and;
2. Schedule tentative engagements against all objectives, in order of priority, with non-conflicting resource profiles.

The first approach is much faster, but the advantage of the second is that one can ensure a maximum number of objectives are engaged. That is, the ordering of schedules can be based on factors supplementary to the objective priority. In PyDES, the first is termed "Just-in-Time" (JIT) scheduling and the second, "Objective-based Priority" (OBP) scheduling.

## 3. IMPLEMENTATION DETAILS

### 3.1 Software Architecture

The PyDES framework is divided into front-end and back-end components. As illustrated in Figure 1, the front-end reads input data in the form of a Scenario data structure, which is then passed to the back-end for simulation. Results are then returned to the front-end for output and display. The front-end can take the form of a script or a graphical user interface. A script allows greater control over the simulation, as well as the definition of additional classes, whereas the GUI tends to be easier to use for those not familiar with the Python environment.

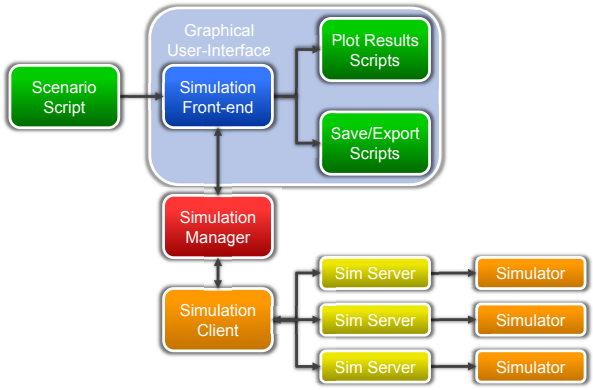


**Figure 1:** Generic Program Flowchart

The Scenario data structure can either be created from another Python script, or written explicitly in the XML or YAML format. It represents the taxonomy of resources and objectives in the relevant application and their attributes in parametric form. Both script and GUI can facilitate modification to the scenario and both can export the results to file in addition to generating interactive graphical output.

The simulation manager is responsible for a number of tasks associated with the execution of the simulation(s). Most importantly, for scenario batches, it distributes and manages the tasks sent to individual simulators and then amalgamates the results returned from each. The tasks comprise, among other fields, the Scenario data structure and parameters controlling the execution of the simulation, such as the number of runs.

When passed to a simulation object, the Scenario structure is replicated with objects representing each resource and objective in place of the dictionary values. Those objects are then allowed to interact at designated times in the event queue. To start the simulation, a 'start' event is fired and an initial interaction takes place, spawning a new set of events. Thereafter, events are fired and spawned as interactions occur repeatedly until no future events exist in the queue.

### 3.2 The Scenario

The Scenario data structure supplied to the PyDES simulation manager and simulator is a Python dictionary consisting of lists of smaller dictionaries within which the resource and objective parameters are stored. Inside the simulation manager, the values for each are used to create the corresponding object.

For the NAD application, the host and generator are represented by Platform and Target objects, respectively. There is also a list of Engagement Managers referenced in the scenario. Resources on each Platform consist of Search Radars, Illuminators (which also represent the Fire Control Radar, FCR), Launchers and an Engagement Manager. The Launchers contain several SAM Types; individual SAM objects are created within the simulator. Each Target consists of a number of Threat Sectors and Engagement Managers. Each of the Threat Sectors contain several ASM Types; individual ASM objects are created within the simulator and are themselves stored inside Objective objects. The skeleton structure is listed below:

```
Scenario = {
  'Platforms' : [ { * ,
    'Search Radars' : [ { * } ],
    'Illuminators' : [ { * } ], # also Fire Control Radar
    'Launchers' : [ { * ,
      'SAM Types' : [ { * } ] # Surface-Air Missile
    } ],
    'Engage Manager' : *
  } ],
  'Targets' : [ { * ,
    'Threat Sectors' : [ { * ,
      'ASM Type' : [ { * } ] # Anti-Ship Missile
    } ],
    'Engage Managers' : [ * ]
  } ],
  'Engage Managers' : [ * ]
}
```

where the asterisk, '*', represents one or more parameter keys and/or values.

In simple terms, the scenario is simulated as follows:

1. The ASM's are detected by each Engagement Manager as they come into the Platform's Search Radar range.
2. The Engagement Manager(s) respond by scheduling a profile of Illuminators, Launchers and a salvo of SAM's against the ASM.
3. When all resource-based events related to the engagement are complete, a post-fire evaluation (PFE) phase is conducted to determine if intercept was successful.
4. If successful, no further action is taken. If unsuccessful, another profile of resources are scheduled against the ASM.
5. The process repeats until all ASM's have either been intercepted or detonated by impacting the Target.

Figure 2 depicts the basic engagement profile for the NAD scenario.

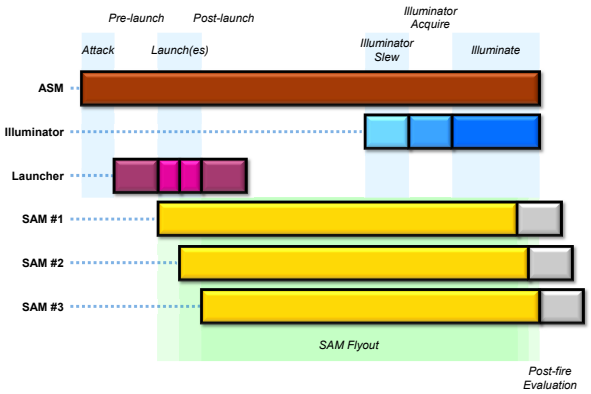


**Figure 2:** NAD Application Engagement Profile

The start-time for the engagement is dependent on the anticipated intercept(s) time, which is determined by the designated range of intercept and the availability of all resources. The availability of each resource is determined by the list of existing/scheduled periods and the start-time and duration required by the engagement. Durations can be static (fixed), such as the Pre-launch delay, or dynamic, such as the Illumination time, which depends on the flyout time, the SAM illumination mode, as well as the terminal guidance duration.

The full time-history for a self-defence (single Target) scenario is shown in Figure 3.

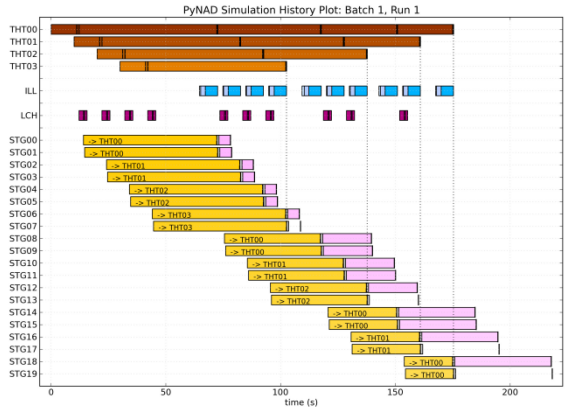


**Figure 3:** NAD Time-History Example

The NAD application has been used to determine many factors related to anti-ship missile defence, including Probability of Survival and Keep-Out Distance for various configurations and threats.

For the UMO application, the host and generator are represented by Platform and Search Area objects, respectively. As with all applications, there is also a list of Engagement Managers referenced in the scenario. Resources on each Platform consist of Autonomous Vehicles (AV's), Launchers, Rechargers, Support Teams and an Engagement Manager. The AV's contain several Sensors and the Support Teams contain a number of Managers, each of which is in charge of several Crew Members. Each Search Area consists of a

number of Track Areas and Engagement Managers. There is a list of Track Paths which define the AV survey paths to be used inside the Track Areas. Each of the SubObject Fields contain several SubObject Types; individual SubObject objects are created within the simulator and are themselves stored inside Objective objects. The skeleton structure is listed below:

```
Scenario = {
  'Platforms' : [ { * ,
    'AVs' : [ { * , # Autonomous Vehicles
      'Sensors' :  : [ { * } ]
    } ],
    'Launchers' : [ { * } ],
    'Rechargers' : [ { * } ],
    'Support Teams' : [ { * ,
      'Managers' : [ { * ,
        'CrewMembers' : [ { * } ]
      }]
    } ],
    'Engage Manager' : *
  } ],
  'Search Areas' : [ { * ,
    'Track Areas' : [ { * } ],
    'Engage Managers' : [ * ]
  } ],
  'Track Paths' : [ { * } ],
  'SubObject Fields' : [ { * ,
    'SubObject Types' : [ { * } ]
  } ],
  'Engage Managers' : [ * ]
}
```

The UMO scenario is simulated as follows:

1. The Track Areas are added to the list of Objectives in order of distance from the Platform.
2. The Engagement Manager(s) respond by scheduling a profile of AV's, Launchers, Rechargers and Support Teams to survey the Track Area with a pre-defined Track Path. The Support Teams consist of a Manager, plus Crew Members with roles as Deck-hand, Launch-hand, Analyst and Operator (if required).
3. During the search phase, any SubObject that the AV passes over and detects is flagged as such.
4. When all resource-based events related to the engagement are complete, including the analysis, an evaluation is conducted to determine if a further reacquire mission is necessary for any of the SubObjects detected.
5. If more data is deemed necessary for some or all of the SubObjects, a reacquire mission is scheduled for them. The reacquire mission has a different profile and Track Path, visiting each of the SubObjects and conducting local searches one-by-one in an order determined by a 'Travelling Salesman' algorithm.
6. The process repeats until all Track Areas have been surveyed and no SubObjects remain to be reacquired.

Figure 4 depicts the basic engagement profile for the UMO scenario.

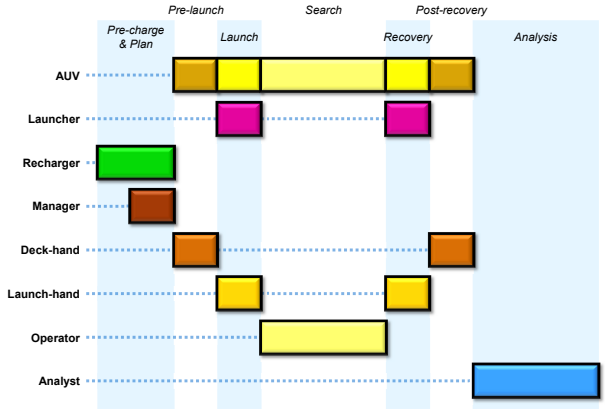


**Figure 4:** UMO Application Engagement Profile

The full time-history for a coordinated survey scenario with two Support Teams is shown in Figure 5. The additional 4 blocks in the Crew Member's state lists represents periods of down-time during which they cannot be scheduled for any engagement.

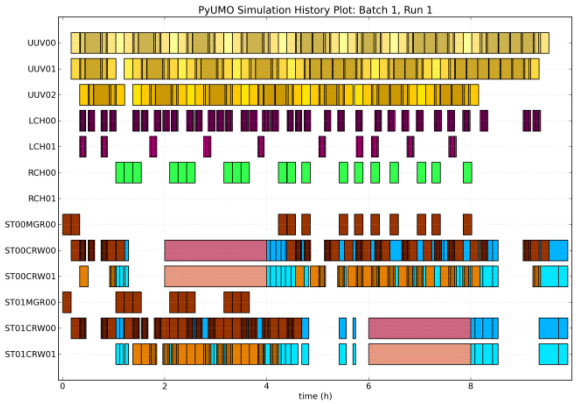


**Figure 5:** UMO Time-History Example

The UMO application has been used to assess differing operating procedures in regard to such factors as survey efficiency, clearance rates and crew workload.

### 3.3 The Simulator

The Simulator defines how the Scenario operates logistically and therefore each application must derive its own version from the base Simulator class. The class attributes consist of a collection of Simulator State-based Objects (SSO's) representing the resources, an Event Queue, Objective List and an Engagement Manager. Class methods include functions for initialisation, simulation and all of the functions that constitute events as well as any supplementary methods involved in the event-driven interactions. The event methods must include 'start' and 'check' operations as a minimum. The 'start' function simply starts the simulation and the 'check' function provides the Engagement Managers with an opportunity to schedule Engagements against any outstanding Objectives.

The most important method of the Simulator object is the 'schedule' function, which inserts states into each of the SSO's state lists corresponding to the engagement supplied. The State List attribute of each SSO is a time-ordered list of states that have been allocated, also known as a state trajectory. At a minimum, each state consists of a start-time, duration, description and identifier. Some SSO's also have a list of references to objects (typically other SSO's) and a start-position and shift. For example, the Illuminator SSO utilises the start-position and shift to represent its initial slew-angle and linear angular displacement. Thus, the state slew-angle displacement can be calculated from the Illuminator's slew-rate property and state duration. Conversely, the state duration can be calculated from the initial slew-angle and displacement. Various functions exist to navigate and manipulate the State List, such as finding states, inserting and deleting states, as well as functions to check for the object's availability, or gaps in its State List.

The Event Queue constitutes a time-ordered list of events to be executed, or fired. Each event is a dictionary object consisting of a start-time, an operation tag, a unique identifier and an arbitrary object (typically an SSO). Unlike the SimStateObject's State List, there is no requirement for a duration to be stored; the event simply indicates that a set of specific actions are to be taken at that time, including the creation of other events.

The Objective List is a priority-ordered list of objectives. Objectives in the Objective List are Python objects instantiated from the Objective class. Objectives have a unique identifier, a list of objects, a detection status flag and an engagement status flag. The priority is defined by the application. For example, the NAD application prioritises threats according to their expected time of impact and consequently attempts to engage them in that order. The default options for detection status are 'undetected' and 'detected'. Those for the engagement status are 'unscheduled', 'scheduled', 'engaged' and 'noengage'. An objective that has been 'detected' cannot be 'undetected' and one that has been 'engaged' cannot be (re-) 'scheduled'.

As with the SSO's State List functions, the Objective List also has functions to navigate and manipulate the list, such as retrieving particular objectives, inserting and deleting objectives, as well as functions to set the detection status and engagement status flags.

The Engagement class holds all resources and the pertinent (State List) utilisation times to engage a specific objective. Engagements are associated with a particular Engagement Manager. The Engagement class encapsulates the scheduling profile for its combination of resources, such as calculating the relative start-times for each resource and determining the minimum time until a valid engagement is possible.

For the NAD application, there are basically two types of limiting conditions: range-limited engagements and; resource-limited engagements. The range-limited engagements can be imposed by launch sectors, engagement sectors and SAM ranges. Resource-limited engagements are imposed by both the Illuminator/FCR and Launcher. The minimum time until valid engagement is determined by searching through the list of times corresponding to limiting ranges and the list of limiting times, as defined by existing active periods in each resource's State List. For each time, the relative start time and duration for each associated resource - the engagement profile - is calculated and then checked for availability. The set of start times corresponding to the minimum can then selected from all valid profiles and used for scheduling the resources, performed by the Engagement Manager in the next step.

For the UMO application, there are only resource-limited engagements, although the number of resources associated with each engagement is much larger, so many more State List times need to be checked in order to find a minimum, valid engagement. Certain resources, such as the Deck-hand, Launcher and Launch-hand are also used multiple times in the one profile, further increasing the number of times checked.

The relative start-times and durations for each resource within a profile can be either fixed or variable. In the former case, the time and duration calculations are straightforward and some efficiencies can be exploited. However, the latter case may depend on any number of factors, such as the resource's current position, or some randomly determined variable, thus requiring several function calls for each time checked.

The Engagement Manager class manages the engagements for all objectives in the Objective List. There are currently two types of engagement policies: Just-in-time and Prioritised scheduling. Just-in-time, or late scheduling refers to the policy of scheduling an engagement against the highest-priority objective as soon as possible. In addition, all remaining (detected), unscheduled objectives are examined for possible engagement and a 'check' event is inserted into the Event Queue at the minimum start time for those. In prioritised scheduling, all detected, unscheduled objectives are cleared and then re-scheduled according to the order of priority in Objective List, each time a 'check' event is fired. However, no 'check' events are explicitly inserted within the engagement process itself. Compared to Just-in-time scheduling, there are both advantages and disadvantages of this policy. The main advantage is that all outstanding (detected, unscheduled) objectives are assessed and engaged at the same time, possibly eliminating any bias which may occur in the just-in-time case by scheduling some objectives at a later time. The main disadvantage is that certain resources can be "locked out" from being employed to engage other objectives by scheduling too early. The difference, albeit subtle, can sometimes result in non-optimal scheduling or, in the worst case, outstanding objectives that are unable to be engaged at all.

### 3.4 The Simulation Manager

The Simulation Manager creates batches of scenarios to be run by the simulator(s). It can run the simulations either in the same, single process, or multiple processes. It can be run as a synchronous (blocking) client, which

sends individual scenarios to any servers as they become available and receives results from them as their simulations are completed. The servers can be run internally, using the Pyro framework, or on the Amazon EC2 cloud, using the PiCloud service.

Scenario batches are defined by parameter lists within the Scenario data structure. Every parameter within the Scenario data structure is assigned a type, such as floating-point scalar, integer, string or list. A parameter space is consequently defined for any value within the struct where a list of such parameters replaces a single parameter. For example, if the floating-point scalar 'speed' parameter of the 'SAM Types' list-element dict was replaced by a list of floating-point values, the parameter space would span that list. If the 'position' two-list (x, y) of the 'Platforms' list-element dict was replaced by a list of two-lists, a sequence of Scenarios with varying platform position would be run.

For the area-defence scenario depicted in Figure 6, the effect of varying the Target position is illustrated with the Probability-of-Survival ($P_s$) heat-map, shown in Figure 7.

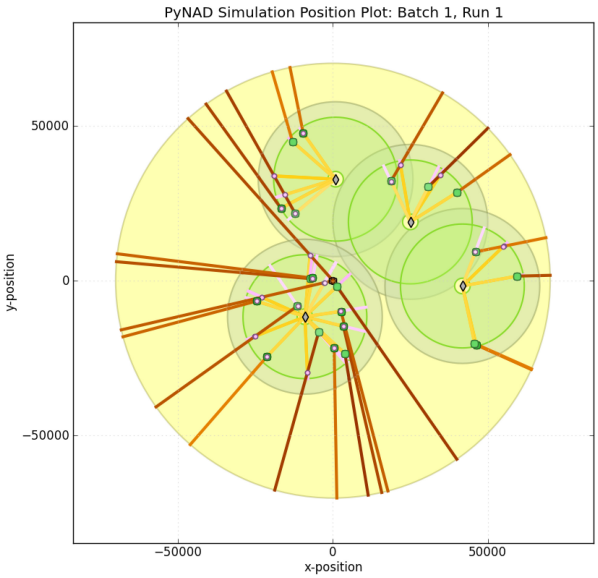


**Figure 6:** Area Defence Scenario Plan View

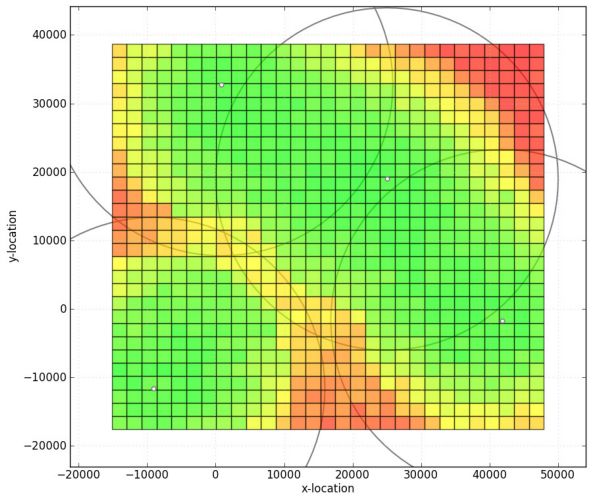


**Figure 7:** Probability of Survival for a NAD Scenario

In the second figure, green indicates a high $P_s$ for the Target and red, a low $P_s$.

Implementing a number of parameter lists within the same Scenario data structure has the effect of defining a complete parameter space, comprising every combination of parameters represented by lists. Due care must therefore be taken to avoid blowing out the size of the parameter space to explore. In the above examples, if we were to define lists of M 'speed' values and N 'position' values, a total of M x N Scenarios would be exercised.

Thus we are left with the problem of simultaneously varying parameters. For example, if the Platform's 'engageCentre' parameter varied according to the Target's 'position' parameter, we would not define two M-lists since that would cause M^2 combinations to be explored. In order to incorporate this in the correct manner, we need to replace one parameter value by either a function or an execution string which produces the desired effect. In the example given, this would entail assigning a function which calculates the Platform's 'engageCentre' based on the Target's 'position' parameter within the same Scenario data structure.

In summary, the Scenario can hold a static parameters, as well as dynamic parameters. The dynamic parameters can be either Batch Variables, which represent a parameter subspace to explore, and Function Variables, which represent a relationship within the data structure.

## 4. COMPUTATIONAL SPEED AND SCALABILITY

When compared with compiled languages, like C++ or even Java, the sole limitation of the Python language is speed. For many applications, PyDES will be fast enough. However, there are various ways in which this shortcoming can be addressed. Namely, by incorporating one or more of the following projects: Numpy, Cython, PyCUDA/PyOpenCL and PyPy.

Numpy is a library for numerical computation and can be used to some extent to vectorise some of the rudimentary list operations. Numpy makes use of optimised linear algebra libraries such as BLAS and LAPACK [7], so is very fast for some operations. Cython [8] simplifies the generation of C++ extension modules for the CPython runtime, by facilitating embedding C code directly in the Python code. However, integration would require some of the functions to be written in C++, which may be too large a task. PyCUDA/PyOpenCL [9] packages enable access to the GPU for accelerated parallel computation, but also require low-level calls to the API and consequently a significant amount of re-coding.

PyPy [10] is an alternative implementation of the Python runtime, which incorporates a JIT compiler for potential speed-up and memory conservation. Of the projects outlined, PyPy shows the most promise, since it is highly compatible with the standard Python implementation. No additional code is required, and no special library calls need be made. Moreover, the first three solutions are targeted at inner-loop optimisation,

whereas PyDES employs many complex outer-loops which would lessen their benefit.

The other approach to increasing performance, and one that has been adopted in the framework, is to make the system scalable. PyDES has options for scaling vertically - by taking advantage of multiple CPU cores - as well as horizontally - by exploiting additional computing instances on the network. Parallel computation is possible with the multiprocessing library, which is now included in the standard Python distribution. Distributed computation is facilitated with the Pyro Remote Objects library. By refraining from using any of the above projects, PyDES has remained a 'pure Python' software package, which yields many advantages: not the least of which is that it continues to be truly independent of operating system and hardware.

Pyro provides a system for employing remote objects over the network. It is very straightforward to construct a client/server architecture for distributed computation. The Pyro library consists of a Name server and a dispatcher, normally deployed on the same 'management' server, as well as a number of workers (servers), deployed on 'processing' servers. The dispatcher is added to the nameserver, then workers are added to the dispatcher when started. In PyDES, the Pyro version of the Simulation Manager contacts the nameserver and receives a list of workers through the dispatcher. It then sends scenarios from the task queue to the dispatcher, which passes the job on to a free worker. As workers return results, they are marked as free for the next task (scenario). Task management is performed by the Simulation Manager to allow the tracking of progress and job control (stop, pause, etc.). This could be done entirely by the dispatcher, but overall progress would be obscured and granular job control not possible.

If the local network resources are insufficient, the commercial PiCloud [11] service can be utilised for distributing the computation. PiCloud employs the Amazon Elastic Compute Cloud (EC2) for high-performance computing and is very easy to set up. In PyDES, the PiCloud version of the Simulation Manager sends the entire list of tasks to the PiCloud server, which then performs the distribution and execution of those. Progress tracking is possible from PyDES, but job control must be performed through the PiCloud web service. As with Pyro, the results are returned as each simulation is completed.

## 5. CONCLUSION

PyDES is a software framework for discrete-event simulation, designed for the analysis of resource based models that are scheduled to meet objectives. The framework has been presented in some detail and includes discussion of the scheduling approach, as well as the software implementation in Python. The Scenario data structure is introduced, along with exemplars from two different applications written within the framework: Naval Area Defence and; Unmanned Maritime Operations. The main classes of the base modules are detailed, including the Simulator, Simulation Manager and Engagement Manager, in addition to the Simulation Objects and queues that hold the events and objectives. In the last section, computational performance is addressed and some features enabling the software to scale are presented.

## REFERENCES


[1] G. van Rossum and F. L. J. Drake, *The Python Language Reference Manual*: Network Theory Ltd., 2011. http://python.org/

[2] W. L. Winston, *Operations Research: Applications and Algorithms*, 4th ed.: Duxbury Press, 2003

[3] T. E. Oliphant, "Python for Scientific Computing," *Computing in Science and Engg.*, vol. 9, pp. 10-20, 2007. http://www.numpy.org/

[4] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, and T. Oliphant, "Numerical Python," Lawrence Livermore National Laboratory2001

[5] J. D. Hunter, "Matplotlib: A 2D graphics environment," *IEEE Computing In Science & Engineering,* vol. 9, pp. 90-95, May 2007

[6] I. de Jong, "Pyro - Python Remote Objects," 4.12 ed, 2012. http://packages.python.org/Pyro4/

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third ed.: Society for Industrial and Applied Mathematics, 1999

[8] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," *Computing in Science Engineering,* vol. 13, pp. 31 -39, March 2011. http://cython.org/

[9] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA: GPU Run-Time Code Generation for High-Performance Computing," *Computing Research Repository,* vol. abs/0911.3456, 2009. http://mathema.tician.de/software/pycuda

[10] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," presented at the Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, 2006. http://pypy.org/

[11] K. Elkabany, A. Staley, and K. Park, "PiCloud," ed, 2012. http://www.picloud.com/