Automation of Inertial Fusion Target Design with Python

Matthew Terry‡*, Joseph Koning‡

Abstract—The process of tuning an inertial confinement fusion pulse shape to a specific target design is highly iterative process. When done manually, each iteration has large latency and is consequently time consuming. We have developed several techniques that can be used to automate much of the pulse tuning process and significantly accelerate the tuning process by removing the human induced latency. The automated data analysis techniques require specialized diagnostics to run within the simulation. To facilitate these techniques, we have embedded a loosely coupled Python interpreter within a pre-existing radiation-hydrodynamics code, Hydra. To automate the tuning process we use numerical optimization techniques and construct objective functions to identify tuned parameters.

Index Terms—inertial confinement fusion, python, automation

Inertial Confinement Fusion

Inertial confinement fusion (ICF) is a means to achieve controlled thermonuclear fusion by way of compressing hydrogen to extremely large pressures (GBar), temperatures (10's keV) and densities (100x solid density). ICF capsule are typically small (~1 mm radius) spheres composed of several layers of cryogenic hydrogen, plastic, metal or other materials. These extreme conditions are reached by illuminating the capsule with a very high intensity (100's TW) driver. This compresses the shell to more than 100 times solid density and accelerates the radially converging shell to very high velocity (300 km/s). As the shell stagnates, a fusion burn wave propagates from a central, low-density, high temperature region to a surrounding high-density, low temperature fuel region. The inertia of the fuel keeps it intact long enough for a significant fraction of the fuel to burn.

There are several approaches to achieving a significant fusion burn, but for this paper consider the shock ignition [Betti2007] approach with the capsule directly driven by lasers. The capsule is a spherical shell of frozen deuterium-tritium ("DT ice"), coated with plastic or another ablator material. The region within the DT ice is filled with DT gas at the vapor pressure. Laser beams directly illuminate the target and deposit energy in the outer most layer called the ablator. The ablation of the ablator supplies the pressure to drive the implosion. We assume a spherically symmetric illumination of the capsule with the total incident power varying in time. The power vs time profile is referred to as the "pulse shape."

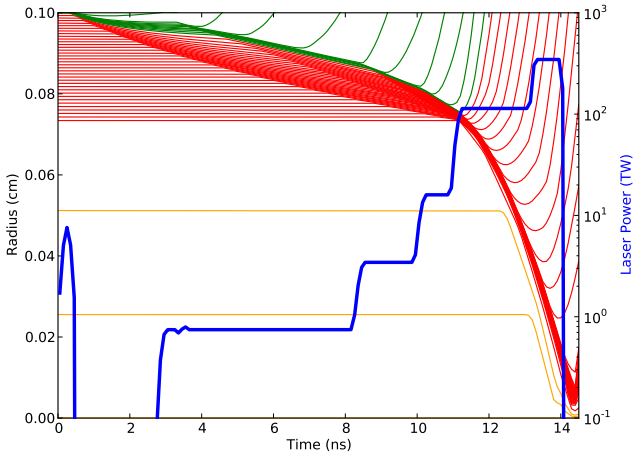

Fig. 1: *A Radius-Time plot of the capsule implosion with the incident laser power overlay. Lines plot the trajectory of fluid particle boundaries. Lines are color coded by material.*

We divide the pulse shape into three logical sections, which correspond to the three phases of the capsule implosion dynamics. The first section is called the "pre-pulse" and is responsible for shock compressing the DT shell to high density. The pre-pulse consists of a short duration, high intensity spike in the laser power (the "picket") and three pedestals, each with increasing laser power. The pre-pulse is followed by the main pulse, which accelerates the shell to moderate implosion velocity (~300 km/s). When the imploding shell stagnates, it forms a central, low density, high temperature hot spot and a surrounding high density, low temperature shell.

The final section of the pulse shape is the igniter pulse. The igniter pulse consists of another pedestal of very high intensity. This section launches a strong shock that arrives just as the shell is stagnating and further heats the hot spot as well as prevents the low pressure shell from coming into pressure equilibrium with the high pressure hot spot. The combination of the stagnation of the shell and the timely arrival of the igniter shock lifts the temperature of the central hot spot above the 12 keV threshold needed to initiate a fusion burn wave. This burn wave propagates into the cold shell where it produces most of the fusion yield.

While restricting our attention to laser shock shock ignition,

* *Corresponding author: terry10@llnl.gov*
‡ *Lawrence Livermore National Laboratory*

there is a lot of potential variability in the composition and structure of capsules and in the pulse shape. Capsule should have sufficient ablator to drive the implosion, but not in excess. Capsule materials must anticipate the effect of fluid instabilities and laser absorption. The capsule should have realistic fabrication tolerances. Laser powers must be set to produce shocks of an appropriate strength and pulse features should be appropriately timed. Additionally, there are several physical processes important in describing an implosion. Due to all of these sources of complexity, ICF targets are designed using sophisticated multi-physics codes, such as Hydra [Marinak1996]. Extensive simulation, helps identify interesting capsule/pulse shapes before resorting to expensive and difficult experiments. The process of designing a capsule is highly iterative, time consuming, interactive process. In this paper we describe the use of and modifications of Hydra to automate significant sections of the target design process. Specifically, we consider the situation where a capsule design and the pulse shape power levels are specified and the timing of the pulse shape is not specified.

When tuning the pre-pulse, we regard the picket as a fixed quantity functioning as a time fiducial for synchronizing the remaining pre-pulse shocks. The picket exists to increase implosion stability by heating the plasma corona, which increase lateral thermal conduction, which in turn smooths out non-uniformities in the deposition of laser energy. These are not effects that can be resolved in one dimensional (1D) simulations, but the picket effects the 1D dynamics and must be included. The pre-pulse pedestals should have their start times set such that their associated shocks reach the gas/ice interface within 50 ps as the picket shock [Munro2001]. Spacing the pre-pulse shocks in this way, prevents them from coalescing in the ice and unnecessarily shock heating the fuel. Also, shocks gain strength with radial convergence, so ensuring that the pre-pulse shocks escape the fuel while it is at large radius helps minimize the shock heating.

The main pulse should be timed to get the maximum fuel confinement for a fixed amount of energy. The appropriate measure of fuel confinement is its peak areal density ($\rho R = \int \rho(r)dr$). Should the fuel ignite, the burn fraction is approximated by $f \approx \frac{\rho R}{\rho R+7}$ [Fraley1974]. Finally, the igniter pulse should be timed so that the target ignites robustly. We implement this as maximizing the fusion yield.

Automatic Tuning

We adopt the general strategy that a tuned pulse can be constructed by serially adding tuned pulse segments. Additionally we require that each property of a pulse segment can be "tuned" by numerically optimizing an appropriately chosen objective function. Our automated pulse tuner ("autotuner") is structured around an iteration over pairs of pulse properties and objective functions. These properties are the start times of the pulse segments and are initially turned off. The tuner iterates through each pulse segment, numerically optimizing it based on its associated objective function. In addition to fixed power levels, the combined energy delivered in the pre-pulse and compression pulse is constrained. The total igniter pulse energy is also pre-determined. It is important to realize that the sequence of properties and choice of objective functions embodies a strategy to achieve the desired target behavior. The automation of this strategy does not guarantee the tuned pulse will produce the desired performance characteristics, just that the design strategy was faithfully executed.

In addition to a sequence of parameters and a definition of an objective function, an autotuing program requires other software infrastructure. It needs to transform parameter values to input files and run directories. The autotuing program needs to gather the appropriate information from a simulation needed by the objective functions. Finally, it needs reasonably efficient numeric optimization routines.

We generate Hydra input files from a Python proxy class that wraps a nearly complete Hydra input file. The proxy has simple pre-processor like capabilities for modifying simple input file statements and for injecting more complicated structures into the input file. For complicated structures, like the laser source specification, it delegates responsibly to special purpose objects. These object follow the convention that `str(obj)` produces a string formatted for inclusion in a Hydra input file. This convention allows objects that define the `__str__()` to lazily evaluate their Hydra representation, while actual strings can be inserted with no boilerplate.

Certain objective functions require very high sampling rates and thus must be run within a running simulation. For this purpose, Hydra has an embedded Python interpreter. Since our tuning program and Hydra's embedded interpreter use the same programming language, it is relatively easy for the control program and Hydra to share data structures. There are two obvious methods: object serialization with the pickle module and object reconstruction using `repr()`. Reconstructed objects are easily modified and more explicit, so we use that method.

All of the optimizations use a simple eight way parallel direct search method. In terms of the number of function evaluations, direct search is less efficient than Newton-like methods, direct search is very inefficient. Typical optimizations requires 32 functions evaluations. Converging to the same tolerance using the BFGS method requires only 12 function evaluations. However, the inefficient direct search method requires only 4 iterations, compared to the 12 iterations with BFGS. We are satisfied with the current performance, but recognize that the use of more sophisticated sampling techniques would likely reduce the number of iterations or the number of parallel function evaluations.

Hydra's Parallel Python Interpreters

Hydra is a massively parallel multi-physics code in use since 1993. The code combines hydrodynamics with radiation diffusion, laser ray trace, and several more packages necessary for ICF design and has over 40 users at national laboratories and universities.

Hydra users set up simulations using a built-in interpreter. The existing interpreter provides access to the program parameters and provides functions to access and manipulate the data in parallel. Users can also access and alter the state while the simulation is running through a message interface that runs at a specific cycle, time or if a specific condition is met.

To improve functionality, the Python interpreter was added to Hydra. Python was chosen due to the mature set of embedding API and extending tools and the large number of third party libraries. The Python interpreter was added by embedding instead of extending Python itself. This choice was made due to the large number of existing input files that could not be easily ported to a new syntax. The Simplified Wrapper and Interface Generator (SWIG) [Beazley2003] interface generator is used to wrap the Hydra C++ classes and C functions.

Users can send commands to the Python interpreter using two separate methods: a custom interactive interpreter based on the

CPython interpreter and a file-based Python code block interpreter. The Hydra code base is based on the message passing interface (MPI) library. This MPI library allows for efficient communication of data between processors in a simulation. The embedded interactive and file based methods must have access to the Python input source on all of the processors. The MPI library is used to broadcast a line read from stdin or a file on the root processor to all of the other processors in the simulation. The simplest method to provide an interactive parallel Python interpreter would be to override the `PyOS_ReadlineFunctionPointer` in the Python code base. This function cannot be overridden for non-interactive processes due to a check for an interactive tty. An alternative interactive Python interpreter was developed to handle the parallel stdin access and Python code execution. For parallel file access the code reads the entire file in as a string and broadcasts it to all of the other processors. The string is then sent through the embedded Python interpreter function `PyRun_SimpleString`. This C function will take a char pointer as the input and run the string through the same parsing and interpreter calls as a file using the Python program. One limitation of the `PyRun_SimpleString` call is the lack of exception information. To alleviate this issue a second method was implemented uses `Py_CompileString` then `PyEval_EvalCode`. The `Py_CompileString` uses a file name or input file information to give a better location for the exception.

The existing Hydra interpreter is the dominant interpreter and must be given control when Python is not in use. The interactive Python interpreter must check for Hydra control commands as well as compiling, executing and checking errors on Python code. The custom interactive interpreter first reads a line from stdin in parallel. Readline support is enabled which gives the user line editing and history support similar to running the Python program interactively. The line is then checked for any Hydra specific control sequences and compiled through the `Py_CompileStringFlags`. If the line compiled with no errors then it is executed using the `PyEval_EvalCode` command. Any errors in compiling or exceptions are checked for a block continuation indicator, syntax error or EOF. Exceptions will be displayed as in Python and available in the output of all the processors.

With the above embedded Python support users can run arbitrary Python code through the Python interpreter. One of the mandates of the effort to embed the Python interpreter was to provide an enhanced version of the existing Hydra interpreter. In order to provide this functionality Python must be able to access the information in the running Hydra simulation. This is accomplished by wrapping the Hydra data structures, functions, and parameters using SWIG and exposing them through the `hydra` Python extension module. The code created by SWIG includes a C++ file compiled into Hydra as a Python extension library and a Python interface file that is serialized and compiled into the Hydra code.

The `hydra` Python module allows users to access and manipulate the Hydra state. Hydra has several types of integer and floating point arrays ranging from one to three dimensional. The multi dimensional arrays have an additional index to indicate the block in the block-structured mesh. The block defines a portion of the mesh on which the zonal, nodal, edge, and face based information is defined, these meshes can consist of several blocks. The blocks are then decomposed into sub-blocks or domains depending on how many processors will be used in the simulation. Access to the multi-block parallel data structures is provided by structures wrapped by C++ interface objects and then wrapped in SWIG using the numerical Python, `numpy`, module to provide the array object in Python.

Users control the simulation by scheduling messages that conditionally execute based on cycle number, time or specific states. These messages can be redefined from Python to steer the simulation while it is running. In addition to the messages, there is a callback functionality that will run a user defined Python function after every simulation cycle has completed. An arbitrary number of callable Python objects can be registered in the code.

Objects in the top level, `__main__`, state are saved to a restart file. This restart file is a portable file object written through the mesh and file I/O library silo [SILO2011]. The Python component of the restart information is a binary string created through the pickle interface augmented with a state saving module. The Python module used for the state saving functionality is the `savestate` module by Oren Tirosh [Tirosh2008]. This module has been augmented with the addition of `numpy` support and None and Ellipsis singleton object support.

Multiple versions of the Hydra code are available to users at any given time. In order to add additional functionality and maintain version integrity, the `hydra` Python module is embedded in the Hydra code as a frozen module. The Python file resulting from the SWIG generator is marshaled using a script based on the freeze module in the Python distribution. This guarantees the modules are always available even if the `sys.path` is altered.

Embedded Diagnostics and Objective Functions

Embedding a Python interpreter within Hydra adds significant capability. One of the first applications was to add a fluid characteristic tracker. Characteristics are eigenvectors of the Euler fluid equations and represent the highest possible signal speed. Characteristics located near a shock, the characteristic will naturally drift toward the shock front or be swept up in int, consequently they can be used to identify the location of the shock front without the difficulty of post processing the moving Lagrangian mesh. The following initial value problem describes the radial location of the characteristic as the flow evolves: $\dot{r} = v(r) - c_s(r)$. $u(r)$ and $c_s(r)$ are the flow velocity and sound speed at the characteristic's current location r. Our characteristic tracker implementation is aware of the pulse shape and starts tracking a new characteristic for each significant feature of the pulse shape. Characteristic positions must be updated every cycle and the tracker is registered as a callback.

Since the tracker is updated every cycle, it is easy to trigger other events based on the behavior of the characteristic. The first use is trigger the simulation to end just after shock breakout time. This is very important as Hydra's only other relevant mechanism for ending the simulation is a maximum simulation time. Burn is explicitly turned off for these scans, so Hydra's burn rate monitor is not relevant. Setting a time limit either leads to under-estimating the shock breakout time and stopping the calculation before gathering important information or setting the maximum time to be very large and wasting many compute cycles. Additionally, we use the location of characteristics to set the frequency Hydra writes output files. Different stages of the simulation have disparate time scales and it is useful to add resolution only when it is needed.

The most important application of the characteristic tracker is producing smooth, non-noisy measurements of the shock breakout time for the shock syncing objective function. To construct a

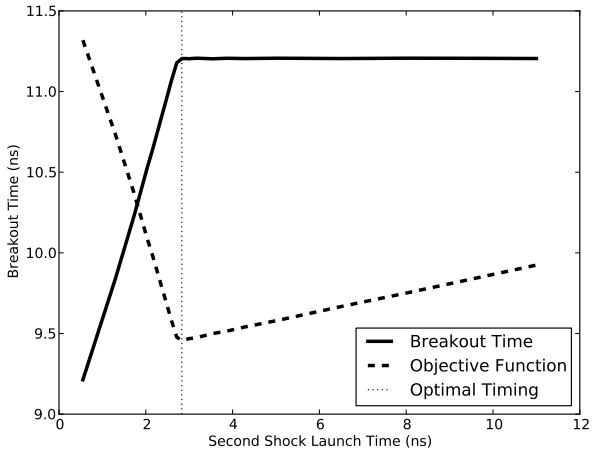

Fig. 2: *Breakout time for a scan of the start time of the second shock. Notice that the objective function minimum accurately locates the inflection point in the breakout vs start time plot.*

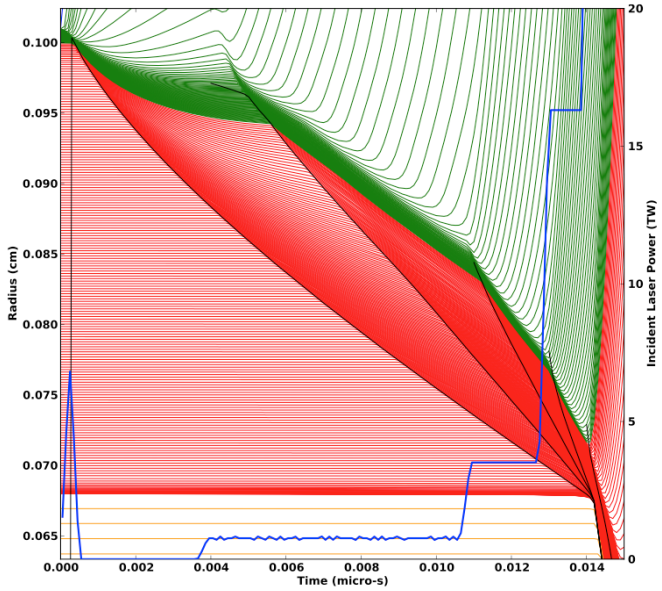

Fig. 3: *R-T plot showing optimal timing of pre-pulse shocks. Shock fronts are identified with black lines.*

shock syncing objective function, first consider the case of two radially converging shocks launched at two different times from comparable radii. The second shock is faster since the wake of the first is warmer and the sound speed is larger. The second shock will eventually overtake the first. If we define a "shock breakout time" as when the first shock enters the gas region, we can plot the shock breakout time as a function of the launch time of the second shock (black line in 2). The appropriate objective function should maximize the breakout time (recognizing that it saturates for large launch times) while also minimizing the launch time of the second shock. We construct an aggregate objective function as a linear combination of the two constraints ($f(t) = \omega t - b(t)$). We find an tuned value of $0.01m$, where m is the slope between the end points of the search region. The parallel direct search optimization method typically converges within four iterations.

Recall from the first section the pre-pulse launches four

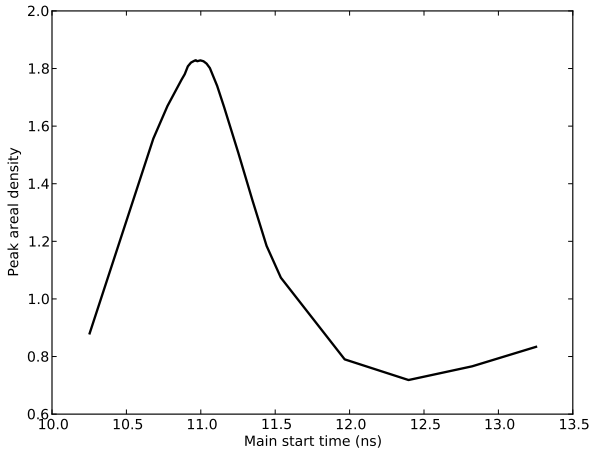

Fig. 4: *Tuning peak areal density*

shocks, all of which should coalesce at the gas-ice interface at the same time. Figure 3 shows the convergence of the pre-pulse shocks well within the required 50 ps tolerance. It should be noted that this shock syncing method only relies on tracking the first shock. Characteristics will sometimes fail to locate the shock if they are located in a region with heat sources that are not sonically coupled to the plasma. Deeply penetrating x-rays, supra-thermal electrons and heavy ion beams are examples. However, it is expected that the ablator and the DT shell should provide sufficient insulation for the picket shock tracker to locate its shock.

Another important embedded diagnostic monitors the fuel areal density (ρR). When tuning the main pulse, the diagnostic monitors the DT ρR, reports the peak value and stops the calculation when the current ρR has fallen to 50% of the peak value. The maximum ρR sets the start time of the main pulse. The igniter pulse start time is tuned by maximizing the fusion yield. Figure 4 shows a peak ρR of $1.8 \mathrm{g/cm}^2$ with a time width of 500ps. Peak ρR is typically found within three iterations. The width in the peak corresponds to mistiming robustness.

Hydra is already well suited for tuning the igniter pulse for maximum fusion yield and needs no additional diagnostics. Hydra monitors the burn rate and has triggers to end the calculation upon completion of burn. Hydra also reports the total fusion yield.

Conclusions

Tuning an ICF pulse to a target is normally a labor intensive, high latency process. We described the desired properties of a tuned pulse and constructed objective functions that will identify the tuned properties. Collecting information for the objective functions requires high frequency sampling of simulation and this data must be gathered within the simulation rather than post-processing a completed simulation. To enable introspective simulations, we add a parallel Python interpreter to Hydra. From these pieces, we constructed a program that tunes a pulse without human intervention. The net result is a significant time savings over manual tuning. Where a typical manual tuning takes several days of attention, an automated tuning takes around 4 hours to execute the same number of simulations.

This work performed under the auspices of the U.S. DOE by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

[Tirosh2008] O. Tirosh, *Pickle the interactive interpreter state (Python recipe)*, http://code.activestate.com/recipes/572213-pickle-the-interactive-interpreter-state/, 2008.

[SILO2011] https://wci.llnl.gov/codes/silo/.

[Betti2007] Betti, R, et al. 2007. Shock Ignition of Thermonuclear Fuel with High Areal Density. *Phys. Rev. Lett.* 98, 155001.

[Munro2001] Munro, David H, et al. 2001. Shock timing technique for the National Ignition Facility. *The 42nd annual meeting of the division of plasma physics of the American Physical Society and the 10th international congress on plasma physics* 8, 2245-2250.

[Fraley1974] Fraley, G S, et al. 1974. Thermonuclear burn characteristics of compressed deuterium-tritium microspheres. *Physics of Fluids* 17, 474-489.

[Marinak1996] Marinak, M M, et al. 1996. Three-dimensional simulations of Nova high growth factor capsule implosion experiments. *Physics of Plasmas* 3, 2070-2076.

[Beazley2003] Beazley,. 2003. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst.* 19, 599--609.