Synesteer Final Report

Peter Stoltz, Doug Dechow, Scott Kruger and Brian Granger

October 15, 2007
## Contents

**Table of Contents**

1 Significance and Background Information, and Technical Approach ............................................ 1

1.1 Identification and Significance of the Problem or Opportunity ........................................... 1

1.2 Background Material ............................................................................................................. 1

1.2.1 Beam Dynamics Application Software ........................................................................... 2

1.3 Technical Approach ............................................................................................................. 4

1.4 Benefits to the Larger Computational Science Community ................................................. 4

2 Phase II Completed Work Items .................................................................................................. 4

2.1 Objective 1 Tasks: IMPACT Code Refactoring ....................................................................... 4

2.1.1 Introduction ..................................................................................................................... 4

2.1.2 Executed Items ............................................................................................................... 4

2.2 Objective 2 Tasks: Forthon Wrapper Modules ....................................................................... 6

2.2.1 Forthon Introduction ...................................................................................................... 6

2.2.2 Forthon Module Wrapping Process .............................................................................. 8

2.2.3 Forthon Wrapped Impact Modules .............................................................................. 11

2.3 Objective 3 Tasks: Python and MPI Integration ...................................................................... 12

2.4 Objective 4 Tasks: Synergia GUI ......................................................................................... 12

2.5 Objective 5 Tasks: User-friendly Toolkit .............................................................................. 13

3 Technical References .................................................................................................................. 18
1 Significance and Background Information, and Technical Approach

1.1 Identification and Significance of the Problem or Opportunity

Particle accelerators play a significant role in the DOE’s near-, medium-, and long-term science programs. In the DOE document *Facilities for the Future of Science* [1], facilities involving particle accelerators comprise 1/2 (14 of 28) of the described activities. In particular for the Office of High Energy Physics (HEP), the International Linear Collider (ILC) and the Muon Collider are top priorities.

Computational modeling and simulation of particle accelerators is an enabling tool for enhancing the full life-cycle of accelerators: analysis, design, optimization, and upgrading. For lower current machines, one can often ignore the interactions between particles (space charge) compared to the interactions of the particles with external fields, and many successful accelerator codes take the approach of discounting space charge. However, for future high-current accelerators like the ILC and the muon collider, space charge is an important effect that researchers must characterize for simulations to be realistic. One approach to including the important effects of space charge is to retrofit the successful accelerator codes that lack space charge by adding a space charge algorithm. For instance, the Synergia code is an attempt to incorporate state-of-the-art space charge models from the Impact code into the Chef accelerator tracking code. One of the issues, however, is that Impact is a Fortran code and Chef is a C++ code, making code integration difficult. A second issue is that neither Impact or Chef has a user-friendly interface to allow non-experts to effectively use the code.

The goal accomplished in this project was to improve the Synergia code by improving the integration of the Impact space charge algorithms into Synergia and improving the graphical user interface for analyzing results. We accomplished five tasks along these lines: (i) a refactoring of the Impact space charge algorithm to make it more accessible by other codes, (ii) development of the Forthon interface between Impact and Python, (iii) implementation of a Python-MPI interface to allow parallel space charge calculation, (iv) a new user-friendly interface for analyzing Synergia results, and (v) a toolkit for doing parallel analysis of Synergia results.

1.2 Background Material

Space charge is suspected of adversely affecting many high-energy accelerators. But, most accelerator computer codes do not include space charge effects because modeling it correctly is difficult and requires significant computational power. Researchers at Fermi National Accelerator Laboratory (FNAL) are developing the Synergia code as an effort to combine state-of-the-art space charge modeling The goal of this project is to develop a state-of-the-art space charge model from the beam physics community into a simulation code for circular accelerators in a highly user-friendly way. In the Phase I portion of this project, we demonstrated the ability to operate Synergia from a user-friendly steering interface, we demonstrated the ability to merge diagnostics with the steering, and we demonstrated the ability to use these new features to model an actual accelerator.
The goal of the Phase II part of this project is to complete the refactoring of the space charge routine, complete the implementation of the automatic FORTRAN 90 wrapping utility, complete the steering modules, complete the GUI interface, and compile these into a user-friendly toolkit for large-scale computation By providing tools for embedding existing codes into a simple scripting language, this work will enable more researchers to use the most sophisticated computer codes and will allow developers to rapidly prototype new computational techniques. This work will also improve the way researchers use large supercomputers by making large-scale computing more user-friendly.

1.2.1 Beam Dynamics Application Software

Recently, there has been significant interest in high-intensity particle beams with quantified limits on particle loss. One result of this interest has been an increase in the need for beam dynamics simulation tools which are capable of modeling space charge effects in particle accelerators. One of the best known tools for modeling space charge effects is Synergia2 [2].

The Synergia2 Beam Physics Framework

The Synergia beam physics framework is an computational accelerator physics application developed at Fermilab for modeling beam dynamics of high-energy accelerators. The application tracks the position and velocity of simulated particles as they move along the length of the accelerator. Synergia models with analytic approximations the various forces on the particles, such as the bending produced by dipole magnets, the focusing produced from quadrupole magnets, and the acceleration produced from cavities. The algorithm for the magnetic force calculations is from the Chef suite developed at Fermilab. Synergia also models in a self-consistent way the repulsive forces between the particles (space charge forces) in three dimensions. The algorithm for these space charge calculations is from the Impact code [3], developed at Los Alamos National Laboratory and presently maintained at Lawrence Berkeley National Laboratory. Few accelerator codes have self-consistent space charge, and even fewer have three-dimensional algorithms.

Like many large, complex scientific software applications, the Synergia [4] accelerator physics framework originated in a non-traditional software engineering environment. A number of the algorithms at the core of Synergia, such as the calculation of magnetic forces on individual particles, are embarrassingly parallel. As a result, Synergia simulations are designed to execute on parallel workstation clusters. Communication between the compute nodes is handled by the Message Passing Interface (MPI) [5]. Synergia is also a hybrid software application that is comprised of software libraries written in two disparate languages: C++ and Fortran 90. Adding to the complexity of the framework is the feature that Synergia simulations are developed and executed in a third language: Python. In short, Synergia, and its most recent incarnation Synergia2, represents the current state of computational physics codes: parallel, multi-language, and complex. An example of Synergia2’s package-level complexity is shown in Fig.1.
Figure 1: Synergia2 is a hybrid code. The primary computational accelerator physics applications are IMPACT and Chef/mxyztlk.
1.3 Technical Approach

Our technical approach in this work was to improve the coupling of the Impact code into the Synergia code. We accomplished this goal by accomplishing five tasks: (i) a refactoring of the Impact space charge algorithm to make it more accessible by other codes, (ii) development of the Forthon interface between Impact and Python, (iii) implementation of a Python-MPI interface to allow parallel space charge calculation, (iv) a new user-friendly interface for analyzing Synergia results, and (v) a toolkit for doing parallel analysis of Synergia results.

1.4 Benefits to the Larger Computational Science Community

A main result of this work is the TxView visualization tool for beam physics applications. We have created this tool specifically to be extendible to any application needing to view remote data, including experimental data, and not just for Synergia results. We hope that other accelerator physics researchers will adopt this tool. The ability to view remote data and develop custom analysis functions, all from a cross-platform GUI, is a unique capability.

2 Phase II Completed Work Items

The tasks listed below will enable us to fulfill the technical objectives described in the original Phase II proposal. The tasks chosen represent the most critical parts of a beam dynamics applications, and matches well to the strengths of Tech-X Corporation as shown below. For each task, we discuss how that completed task satisfied the technical objectives of the proposal.

2.1 Objective 1 Tasks: IMPACT Code Refactoring

2.1.1 Introduction

It is often desired to extend existing software so that it can be used in situations for which may not have originally been designed. Unfortunately, software often resists our efforts to change it. One set of techniques that has been developed for making deliberate, incremental changes to software is refactoring [6].

In the Phase I portion of this project, the IMPACT beam dynamics library was given a surface-level refactoring that, primarily, flattened the available namespace. This resulted in a a large, global variable space. Additionally, the resulting application had a clunky, file-driven interface. A graphic depicting this scenario is shown in Fig. 2.

2.1.2 Executed Items

One of the the initial goals of this Phase II project was to create a set of Fortran “components” that we could programatically manipulate as objects. This goal requires solid, principled, object-oriented techniques such as encapsulation and clean, separable interfaces. In order to meets these aims, it was necessary to perform a second refactoring of the code that was created in the initial refactoring.
Because of the complexity inherent in a complete refactoring of a mature Fortran code, we made a pragmatic decision to complete a secondary refactoring that would consist of the following tasks:

1. parameterization of “empty” subroutines and functions
2. cleaning up of the global namespace
3. renaming of poorly named variables

These efforts were focused on extending and deepening the work that had been done to create the AccSimulator Fortran 90 module in the Phase I project. The work proceeded in three separate activities. The activities and the resulting software artifacts, primarily in the form of Fortran 90 user-defined types, are listed below.

1. Migration of Common Blocks
   (a) GlobalData
   (b) OtherData

2. Creation of Initialization Subroutines
   (a) AlphaOmega, primarily contains data
   (b) RoundPipe, a collection of subroutines

3. Separation of Space Charge Subroutines
   (a) Maps, contains the CheckError functions
(b) SpaceCharge, the reason this all was done

Additionally, the following activities were also performed. The files Input.f90 and Output.f90.cpp, a generated file, required significant changes to work with the new subroutines. As an example, the interface to *in2Input* was reparameterized.

Finally, new functionality, including defining and top-level scoping for the primary data structures, beam bunch and beam line, was added to main.f90. A graphic that gives a macro-level look at the resulting, refactored, renamed Synergia2 computational accelerator physics framework is shown in Fig. 3.

### 2.2 Objective 2 Tasks: Forthon Wrapper Modules

In order to make the necessary Synergia Fortran 90 modules accessible via Python, we followed a tool-based methodology to generate Python extension modules. For the task, we used a Python-based code-generation tool, Forthon [http://hifweb.lbl.gov/Forthon](http://hifweb.lbl.gov/Forthon) (created by our colleague Dave Grote of LBNL), to automate the module wrapping. The Forthon-generated extension modules were then compiled via our autotools-based [7] build process.

We employed an exploratory programming development model in an effort to define only those interfaces that were necessary to develop accelerator simulations. This involved the creation of a series of prototype simulations. The guiding principle was to develop a minimal interface for each needed Fortran 90 module.

The end result of our software build process was a set of shared libraries (*.so files on Linux) that can then be imported into Python. Once the wrapped Fortran 90 modules have been imported into the Python interpreter, they can then be used for testing and further prototyping. Additionally, the wrapped modules are also suitable for incorporation into Python scripts.

#### 2.2.1 Forthon Introduction

The canonical Python language extension tool is the *Simplified Wrapper Interface Generator* (SWIG) [8]. SWIG works by ingesting a user defined *interface file*. The interface file defines the data types and function signatures of the module that is targeted for wrapping. In the case of a C language module, this equates to what is contained in a header file. In fact, oftentimes a SWIG interface file is simply an included C-language header file. SWIG takes the interface file and generates the necessary C interface code for a Python module.

Forthon takes this approach but adds an additional step where it generates proxy Fortran stubs. The C code generated by Forthon is essentially a set of proxy functions that can be called from Python. The C-based proxy function can then invoke corresponding Fortran functions and subroutines. Additionally, Forthon has the ability to create new Python types that mirror Fortran user defined types. This was the feature that attracted us to Forthon as a tool. The Impact source code makes heavy use of Fortran90-based, user defined types. An example of using Forthon to wrap an Impact user defined type is provided in the following section.
Figure 3: The resulting Synergia2 computational physics framework.
2.2.2 Forthon Module Wrapping Process

Using the Forthon tool to automatically generate wrapper code for Fortran90 modules and user defined types is a well understood process. The four steps that are necessary to complete the process are enumerated below:

1. Define the Synergia module: this is related to the refactoring activity;

2. Create the Forthon Interface Definitions file: a Forthon readable file, ending in .v that describes the user-defined Synergia types and subroutines;

3. Create Proxy Functions: call-throughs for the targeted Synergia subroutines; and

4. Run the Forthon tool: generate the libraries contained the Python extension modules with –nowritemodules option.

Each step in the overall wrapper generator activity will be discussed in separate section.

Impact Module Modifications

The Fortran code refactoring that was necessary to create an appropriate level of encapsulation for the Impact space charge routines is described in section 2.1. A similar process was undertaken for each of the Impact modules that were wrapped using the Forthon tool. A complete list of wrapped Impact modules is provided in section 2.2.3.

After the refactoring was performed, a final Fortran-related step was necessary to make the Impact modules callable from Python. Forthon requires the addition of a private, integer variable named cobj to the type declaration of each user-defined type targeted for wrapping. The type declaration for a BeamBunch is used to demonstrate this technique. It is shown below.

type BeamBunch
  ! private
  integer(kind=4): cobj__ ! Needed for Python reference counting
  double precision : Current,Mass,Charge
  integer : Npt,Nptlocal
  integer : maxid
  double precision, pointer, dimension(:,:) : Pts1

  !reference particle
  double precision, dimension(6) : refptcl
end type BeamBunch

The inclusion of the cobj__ integer variable is necessary for Python-side reference counting to function correctly.
**Create the Forthon Interface Definitions File**

Forthon is a code generation tools for the domain of language interoperability. In fact, Forthon’s domain is even more restricted in that it only connects Python to Fortran. Like most language interoperability tools, Forthon depends on an intermediate description of the module types that are to be wrapped in the target language. Again, like many of the tools that live in this conceptual space, Forthon makes use of an interface description language to describe the module types in the target language.

Forthon is able to work Fortran modules that contain both intrinsic and user defined types. The Forthon distribution contains a number of sample programs that demonstrate the use of its interface description language. That Forthon interface description that was used to wrap the Impact *BeamBunchModule* is shown below.

```
BeamBunchPkg

***** BeamBunchModule:

%%%%% BeamBunch:
Current real
Mass real
Charge real
Npt integer
Nptlocal integer
maxid integer
refptcl(6) real
Pts1(::,:) _real

***** Subroutines:
construct_BeamBunch_external(thisbeambunch:BeamBunch,
   incurr:real, inkin:real,
   inmass:real, incharge:real,
   innp:integer, phasini:real) subroutine

inject_BeamBunch_external(this:BeamBunch,
   injected:BeamBunch) subroutine

Notice that the subroutines that are used to provide access to the Impact *BeamBunch-module* are also described in the interface file (*BeamBunch.v*). The subroutines also need proxy subroutines to function as intermediaries, and those functions are described in Section [2.2.2]

**Create Proxy Functions**

Forthon was originally intended to be used in a manner such that it was also used to generate Fortran implementation code in the form of subroutine stubs. This code, of course, already exists in Impact. In order to make it work with the existing Impact Fortran modules, it is
necessary to allow Forthon to generate proxy subroutines for the extant Impact subroutines. In this context the proxy subroutines act merely as call throughs. The Forthon generated proxy subroutines for the Impact *BeamBunchmodule* are shown below.

```fortran
SUBROUTINE construct_BeamBunch_external(thisbeambunch,incurr, inkin,inmass,incharge, innp,phasini)
  USE beambunchmodule
  type (BeamBunch), intent(inout) :: thisbeambunch
  double precision, intent(in) :: incurrr,inkin,inmass,incharge,phasini
  integer, intent(in) :: innp

  CALL construct_BeamBunch(thisbeambunch,incurr, inkin,inmass,incharge,innp,phasini)
  RETURN
END SUBROUTINE construct_BeamBunch_external

SUBROUTINE inject_BeamBunch_external(this, injected)
  USE beambunchmodule
  type(BeamBunch), intent(inout) :: this
  type(BeamBunch), intent(in) :: injected

  CALL inject_beambunch(this,injected)
  RETURN
END SUBROUTINE inject_BeamBunch_external

SUBROUTINE BeamBunchdealloc()
  !TYPE(BeamBunch),target:: oldobj__
  !TYPE(BeamBunch),pointer:: poldobj__
  !integer:: error
  !poldobj__ => oldobj__
  !DEALLOCATE(poldobj__,STAT=error)
  !if (error /= 0) then
  !  print*,"ERROR during deallocation of BeamBunch"
  !  stop
  !endif
  RETURN
END SUBROUTINE BeamBunchdealloc
```

One of the reasons that it is necessary to use these proxy subroutines is exemplified by the subroutine *BeamBunchdealloc()*. As the subroutine’s name alludes to, this subroutine is responsible for deallocating instances of the user-defined type *BeamBunch*. Other such memory-management tasks can also be carried out by the Forthon defined proxy subroutines.
Run the Forthon tool

Forthon uses the Python distutils system to control its own build process. Support for the various Fortran compilers that can be used with Fortran is implemented in the Python class `FCompiler`. Adding support for a new compiler is described in the documentation for `FCompiler`. The task depends upon adding a new method to `FCompiler` that makes use of the following naming scheme: `machine_compiler()`. An example of this is the method: `macosx_g95()`.

It was necessary to integrate the Forthon build system into the existing Autotools-based system that is used by Impact. In order to do this, it was necessary to define new variables in the files `Makefile.in` and `Makefile.am`.

The `Makefile.in` file additions are shown below.

```makefile
ft_Bdir = $(abs_top_builddir)/Forthon_Interfaces
ft_Sdir = $(abs_top_srcdir)/Forthon_Interfaces
ft_Tdir = @FORTHON_TEMP_BUILDDIR@
```

The `Makefile.am` modification was tailored to use the g95 compiler.

```makefile
##FORTHON=Forthon -F @FC_BASE@ --fargs @FCFLAGS@ --nowritemodules
FORTHON=Forthon -F g95 --nowritemodules
```

### 2.2.3 Forthon Wrapped Impact Modules

In order to create Synergia2-based beam dynamics simulations, it was necessary to wrap the following Impact modules and user defined types with Forthon:

1. BeamBunch — the particle store;
2. CompDom — the domain decomposition;
3. FieldQuant — the field;
4. Pgrid2d — the processor communication grid; and
5. SpaceCharge — the modularized Impact space charge routines.

Each of these modules can be loaded into a Python interpreter and manipulated as if it were a Python object. An section of Python code from a Synergia2 simulation, `open-channel.py`, is shown below. The code demonstrates creating and initializing an Impact `BeamBunch` user-defined type.

```python
b_impact = bunch.Bunch(current, bp, num_particles, pgrid)
b_impact.generate_particles()
```
2.3 Objective 3 Tasks: Python and MPI Integration

The primary Synergia2 libraries, Impact and Chef, both a parallel applications. Each library was designed to work with the Message Passing Interface (MPI) library. In order to continue to distribute our simulation calculations across cluster-based environments, it was necessary to find a solution that would allow us to do this with Python. There are a wide variety of Python-MPI software systems. In the course of this project, we evaluated three different systems: PyPar [9], pyMPI [10], and mpi4py [11].

PyPar is a very minimal implementation of an MPI binding. It was originally developed for pedagogical purposes. Examination of its source code reveals that it is largely a hand-rolled, SWIG-like, Python type extension module. PyPar ultimately proved to be unsuitable because of PyPar’s difficulties with guaranteeing a correct MPI initialization sequence. The MPI initialization process is quite sensitive to the start-up state of the underlying software environment, and PyPar is known to have difficulties negotiating the correct start-up sequence.

The pyMPI library is a significantly more complete Python-MPI binding. The pyMPI library provides full support for the MPI-2 standard. Unfortunately, the PyMPI implementation of the gather() routine proved to be unworkable with our early prototypes. Eventually, we were able to succeed with mpi4py. The mpi4py toolkit is part of a larger set of tools that were specifically developed to be used with PETSc-based, Python-steered cluster simulations. The mpi4py bindings are based on work done with Object Oriented MPI (OOMPI) [12]. OOMPI was essentially a C++ library for performing MPI calls. We have been using mpi4py for about twenty-four months, and thus far, it has worked well.

An example of how mpi4py is used in a Synergia simulation is shown below:

```python
if MPI.size == 1:
    proccol = 1
num_particles = adjust_particles(griddim[0]*griddim[1]*griddim[2] *
                                     part_per_cell,MPI.size)
```

The above code sample is used in Synergia2’s load-balancing activity.

2.4 Objective 4 Tasks: Synergia GUI

A main goal of this work was to develop an easy-to-use interface to allow researchers to productively use the Synergia code without needing to be computing experts. We wanted this interface to have three features: (i) the interface should work on Windows, Mac, or Linux, (ii) users should have seamless access to remote data, such as data from a supercomputing center and (iii) users should be able to interact with data rather than choose only prescribed views. We also wanted our solution for this problem to be usable with other simulation codes, not just Synergia.

We call our interface solution TxView. TxView is written in Python to make it freely available to any user. Further, it is built on top of the Qt windowing toolkit, and so works with Windows, Mac, and Linux. TxView uses the Data Access Protocol (DAP) to allow user-friendly access to remote data. Finally, TxView allows user-defined functions, so users can interact with their data in a customized way.
Figure 4: The TxView application is the main final result of this work. TxView provides an easy-to-use, interactive way for researchers to analyze simulation results. For instance, this figure shows momentum phase space for a particle beam.

A main design decision with the Python architecture of TxView was to support an interactive command line Python session simultaneous with the GUI. This allows users to interactively analyze data in a customized way at the Python prompt while at the same time using the GUI features. Combining interactive, command line analysis with GUI-based analysis is a unique feature of TxView.

Another main design decision was to support seamless browsing of remote data sets (data that resides on a cluster, for instance). We show the remote data browsing capability in Figs. 4 - 8. The window on the left is a file browser window, and both local and remote files appear in this chooser.

The TxView tool is our main deliverable result for this Phase II work. We believe this tool will work for any DoE researcher needing to view particle or field data, including data from experiment. We hope the TxView product becomes widely used in the DoE research community.

2.5 Objective 5 Tasks: User-friendly Toolkit

Synergia2’s choice of Python as a language to steer computations opens the door for exploring ways that parallel calculations can be made interactive in the same way that Mathematica and Matlab are interactive. This type of interactivity becomes even more important when large distributed data sets need to be analyzed and visualized.

Our last task for the phase II project involved exploiting interactive calculations with Synergia2. These capabilities are being provided by the open source project, IPython [http:]
Figure 5: The TxView application allows researchers to view multiple particle species simultaneously.

Figure 6: TxView also allows easy visualization of field data. This figure shows the fields from an accelerating cavity structure.
Figure 7: Slider bars allow the user to explore details of the fields.

Figure 8: The TxView application includes three-dimensional graphics rendered using VTK.
IPython is an enhanced interactive Python shell that has become the de facto shell for interactive scientific computing in Python. Over the past two years, the IPython developers have implemented a robust architecture that allows parallel programs to be developed, debugged, testing and executed interactively. Our current work in this area is focusing on integrating IPython with Synergia2. In particular we are developing a set of tests, demonstrations and usage cases to see if Synergia2 would benefit by using of IPython. In one of our early tests, we were able to use IPython to interactively analyze and visualize multi-Gb HDF5 files of particle data on a remote cluster. Additionally, IPython has been used for interactive analysis of data produced by Synergia2. An example of an interactive diagnostic session is shown in Fig. 9.

Other data analysis tools have also been integrated into the IPython-based, User-friendly Toolkit that was developed during this phase II project. One such tool is PyLab. An example of the kind of data plotting that can be performed by PyLab is shown in Fig. 10.

http://matplotlib.sourceforge.net/
Figure 10: Results of computing the beam envelope equation using PyLab.
3 Technical References

References


