

**The Arroyo Project:
C++ Libraries for Simulations of Wave Propagation
through Turbulence
DRAFT COPY
WARNING, SOME FUNCTIONALITY
DESCRIBED IN THIS DOCUMENT NOT YET
CHECKED INTO CVS**

Dr. Matthew Britton
Caltech Optical Observatories

April 2, 2003

Adaptive Optics

Two years ago the National Research Council (NRC) declared a giant segmented mirror telescope (GSMT) to be one of two key projects in its decadal survey of ground based optical and infrared astronomy (Dressler *et al.* 2001). The NRC specifically identified adaptive optics (AO) as one of the critical components necessary to achieve the science goals of a GSMT. Extremely high precision adaptive optics systems have the potential to enable direct detection of extrasolar planets in the infrared, while such systems may be used to achieve moderate correction in the visible. Extending the adaptive optics correction using laser beacons will permit AO observations of science objects over the entire sky. These systems are known as multi-conjugate AO (MCAO) systems, because they require the use of multiple deformable mirrors conjugate to different altitudes. Taken together, these types of AO systems are poised to revolutionize the field of observational optical and infrared astronomy.

The adaptive optics systems that will enable such science will be extremely challenging and expensive to construct. These systems require substantial investments in the development of components such as high power lasers, high actuator count deformable mirrors, and fast wavefront sensors. Considerable evolution in the capabilities of real time computing systems will be necessary to calculate and apply the AO correction. Many design problems have been identified that must be mitigated before these systems would even operate, and there are almost certainly other problems that have not. Given the level of uncertainty in the operation and cost of such AO systems, it is worthwhile to consider ways to balance the costs and minimize the risks. An analysis of the expected performance of a candidate AO system would obviously be of key importance.

A theoretical calculation of the expected behavior of the system is obviously the first choice for performance analysis. This approach has worked quite well for the current generation of AO systems.

Error budget estimates appear to agree with measured system performance to a reasonable degree of fidelity (Troy *et al.* 2000). Such calculations rely on the assumption that sources of error in the AO system are independent. For the “extreme” AO systems that will be used for planet detection, the requirements on fidelity become so severe that it is difficult to anticipate what new sources of error will actually become important. Second order interactions between different sources of error may need to be incorporated into the analysis as well. The operation of an MCAO system actually relies on the existence of subtle correlations between measurements from different guide stars. Correlations like these can make a full analysis hopelessly complex. Another effect of potential importance is that AO systems aim to compensate for an inherently random process. Analytical predictions may be made only for statistically averaged quantities, and so cannot describe how the system actually operates on short timescales. Finally, in the next generation of AO systems external measurements will probably be introduced to improve the operational performance of these systems. This becomes an added difficulty in effecting an analysis of the system performance.

Numerical simulations can help to test concepts when analytic predictions appear intractably difficult due to the complexity of interactions in the system. The advantage of a simulation lies in the fact that absolute truth is known throughout the entire system. In our case, the system can be regarded as consisting of the science field, the atmosphere, the telescope and AO system, the science instrumentation and the data analysis algorithms. For the new AO systems that we hope to construct in the next decade, numerical simulations may be used to predict system performance. This would allow us to optimize the design of such systems by evaluating the cost versus performance improvement that results in changing parameters of the design. To actually trust the results of such simulations, one would like to test their predictions against the current generation of AO systems. One can also compare the statistically averaged results from sufficiently idealized simulations against known analytic results. I describe many tests like these in the section on verification below.

Simulation Applications

In this section I would like to briefly mention a number of applications for simulations in general, without limiting the discussion to any particular simulation. These applications are summarized in Table 1.

Several experimental techniques aim to provide data on atmospheric properties. In Scidar one traditionally attempts to map the vertical turbulence distribution using pupil plane observations from two stars at small angular separation. Some researchers have suggested using the data from a Shack-Hartmann wavefront sensor as a Scidar-like instrument to provide real-time estimates of the turbulence profile for use by the AO system. Lidar observations are used to measure other properties of the atmosphere. The most important Lidar measurements for AO are used to map the strength, evolution, and vertical distribution of the sodium layer. A Generalized Seeing Monitor (GSM) can provide measurements of the outer scale of turbulence. It is possible that all of these experimental techniques will be used in the operation of future laser-based AO systems, and that the experimental data may be used by the AO system in affecting its correction. Since all of these experiments rely on electromagnetic wave propagation through the atmosphere, there is an opportunity for numerical simulation to play a role in establishing the precision of these techniques. In particular, one can compare predictions from simulated Scidar and GSM observations to the known properties of the atmosphere used in the simulation.

Scientific instrumentation must operate in the presence of partially compensated or uncompensated atmospheric turbulence. Two examples of particular importance for the next generation of AO systems

Application

- Scidar
- Lidar
- Generalized seeing monitor
- Scientific instrumentation (e.g. coronagraphs, integral field spectrographs)
- Data reduction algorithms (e.g. blind deconvolution)
- External thermal and vibrational influences
- Investigation of AO system performance and tuning
- Reconstructor testbed
- Wavefront sensor testbed
- Segment phasing algorithms
- PSF's for segmented mirror telescopes
- Phase diversity
- Extreme AO performance predictions
- MCAO performance predictions
- Optical/IR interferometric performance predictions

Table 1: Applications that may benefit from simulations. Arroyo aims to provide much of the underlying functionality necessary to realize these simulations.

are coronagraphs and integral field spectrographs. Coronagraphs are designed to suppress light from a bright star so that a very deep image may be made of the surrounding region. Integral field spectrographs are designed to divide a spatially extended object into small pieces and produce a measured spectrum for each piece. The performance of these instruments may be seriously impacted by residual errors that are not corrected by the AO system. These instruments may also be operated on short enough timescales that the resulting data is not completely statistically averaged or on timescales long enough that the properties of the atmospheric turbulence changes over the course of the measurement. For a given atmosphere, AO system and piece of instrumentation, simulations can help to indicate the expected performance of the whole system. This would be useful in evaluating different instrument designs.

Data reduction algorithms provide another possible application for simulations. Examples of postprocessing algorithms include the Richardson-Lucy deconvolution algorithm, blind deconvolution algorithms used by IDAC, and field-dependent deconvolution algorithms. These algorithms generate predictions for the PSF, which may be used to derive astrometric and photometric properties of sources in the field. PSF determination from wavefront sensor data provides an example of a useful real-time algorithm. In simulation the absolute truth for the entire system is known, including all PSF's, absolute astrometry and photometry for all sources, and simulated telemetry data. We can use these known properties to test the predictions made by these algorithms.

External influences on the telescope and AO system such as wind shake, differential thermal expansion and gravity deflection can degrade performance if the system design does not account for them. This particular area of simulation is highly evolved, as it is used in so many applications.

AO systems in operation today often have a small number of parameters that may be tuned during observations. These typically include the control loop gain and rate, and possibly a selection of reconstructors. Tuning these parameters based on the perceived atmospheric conditions and guide star properties often seems more of an art than a science, even for such a small parameter space. For future

AO systems it is to be expected that many additional tunable parameters will become available to the user. For instance, one has the freedom in an MCAO system to choose an LGS asterism that optimizes the quality of a science observation. The choice of asterism could be quite different depending on whether the observer wishes to optimize the AO correction over the whole field, or at a small number of locations in the field. Ultimately one would like to be able to automatically change the operational characteristics of the AO system based on the character of the science observation and in response to changes in the measured atmospheric properties. The controlled environment of a numerical simulation provides a much better means of testing system tuning techniques than trial and error at a telescope, because simulations may be rerun repeatedly using the same atmosphere but varying the system parameters. Elaborating on the example above, it is interesting to speculate on whether it would be useful to run simulations of particular science fields in advance of actual observations so as to be able to make educated real-time decisions concerning observing strategy.

There is a very large literature on the creation of reconstructors for both single and multiconjugate adaptive optics systems. The most sophisticated types of single conjugate reconstructors attempt to use information on the statistical properties and temporal evolution of the atmospheric turbulence to enhance the quality of the correction. Other reconstructors have been designed for optimal computation speed so as to decrease the real-time computational requirements. For MCAO systems, reconstructor research is in a very early stage. Comparative analysis of reconstructor performance is difficult to carry out both theoretically and experimentally. With the former approach it is difficult to model the manner in which reconstructor properties ultimately affect the PSF. In the latter approach, one unfortunately has no control over the temporal evolution of the atmospheric turbulence, and direct comparisons of observations taken with different reconstructors are complicated by this evolution. Numerical simulations provide absolute control over the input conditions, and so provide an extremely useful testbed for reconstructor evaluation.

Similarly, a large variety of wavefront sensors have been proposed for AO. These include the Shack-Hartmann wavefront sensors and curvature sensors currently used in a number of AO systems. There are also novel proposals for wavefront sensing, which include pyramid sensors and other exotic approaches (van Dam & Lane, 2002). Numerical simulations may be used to compare the performance of AO systems that employ different wavefront sensing schemes.

Calibration is another area in which simulations may play a role. Phasing the individual segments of a GSMT is expected to be a challenging problem, and the wave optics simulation can provide comparative tests of algorithms and can indicate expected levels of residual error. Simulating the PSF of a segmented mirror telescope is another useful application. Finally, phase diversity provides a technique for calibrating non-common path errors in an AO system. For this algorithm to work, a realistic representation of the systems and the ability to propagate wavefronts through this system are required. Calibration applications like this require only a subset of the functionality necessary for performing simulations like the ones described above for single conjugate AO systems.

Compared to the current generation of single conjugate AO systems, the AO systems currently envisioned for detecting planets in the infrared will require exquisitely fine levels of control. Development of these systems will likely require a constant and persistent battle against numerous sources of systematic error. Consequently these systems will go through several stages of refinement before the necessary precision can be reached. As each systematic is identified and removed, the precision will increase until the next source of systematic error is encountered. Nevertheless, the very compelling nature of the science case for extrasolar planet detection clearly makes such an effort worthwhile. The identification of sources of systematic error is a real art, as the signature of the error must be identified with subtle interactions in the AO system. Numerical simulations can play a role in the identification of systematic errors, since

these subtle errors may be investigated under tightly controlled conditions.

The construction of MCAO systems for GSMT's will be every bit as challenging as constructing AO systems for planet detection. The MCAO system proposed for the 8 meter Gemini telescope will serve as an incredibly valuable test of the MCAO concept. However, the difficulty in performing MCAO correction grows with telescope diameter, so that MCAO on a GSMT will likely be substantially more difficult. In fact, a self-consistent proposal for a laser-based MCAO system on a GSMT has yet to be made. A number of problems still exist that must be solved if these systems are to be able to operate. These include the effects of finite LGS spot size, the necessity of supplemental natural guide stars to provide information on low order modes to which the laser beacons are insensitive, and the challenging and complex problem of MCAO reconstruction. Numerical simulations may provide the only method for predicting the performance of proposed MCAO systems on GSMT's.

It is interesting to note that for the extreme AO and MCAO systems above, the effects of high altitude turbulence can become much more important than in the case of AO systems in use today. In the current generation of AO systems, high altitude turbulence contributes a very small component of aberration because of its relatively low density. While this turbulence does give rise to off-axis degradation of the science field due to anisoplanatism, there is nothing one can really do about this in single conjugate AO. For high precision AO systems used for planet finding, scintillation may play a limiting role in the stability of the PSF. Scintillation arises when phase fluctuations imprinted on the wavefront mix into amplitude fluctuations due to free-space propagation. Since the high altitude turbulence is encountered first, scintillation preferentially arises from this material. For MCAO systems, high altitude turbulence gives rise to the anisoplanatic error that the system aims to compensate. High altitude turbulence is typically moving at higher velocity, and is correspondingly more difficult to correct.

Another astronomical application for numerical simulations of wave propagation in turbulence exists in the area of performance prediction for optical and infrared interferometry. Most of the considerations above translate directly to an interferometric system, with the added complication that the interferometric system must be simulated as well. It is expected that this application may become important as the interferometric systems on Keck and the VLT continue to evolve.

Finally, there are a number of other applications that are not directly related to astronomical adaptive optics, but that have a great deal of overlap. Examples include optical communication links or imaging applications in which signals must traverse the atmosphere. These applications have themselves spawned a number of simulation efforts, most of which are classified and/or proprietary.

Review of Some Simulation Tools

There are a large number of simulations that contain functionality relevant to some of the applications described above. It is an extremely difficult task to attempt to review the status of these simulations and their relevance to the problems of interest in AO. Part of the problem is the lack of documentation for these projects. Another problem is the proprietary nature of most of the software. Nevertheless, I will attempt to briefly describe some of the other projects with which I am familiar. For several of these projects some sort of documentation is available on the web. URL's for these sites are listed at the end of this paper. I expect that I have made several misrepresentations and omissions in this section, and hope that people reading this document will fill me in.

Perhaps the most general, widely disseminated tool for optical simulations is the commercial product Zemax. Zemax performs geometric ray tracing simulations, though some support for diffractive propagation of wavefronts was added in the last release. Zemax contains no method for modelling the

turbulent atmosphere. Zemax is written in C, is available only for Windows, and to my knowledge has no parallelization capabilities. Zemax has a robust, commercial-grade graphical user interface.

Macos is a simulation written by Dave Redding and others at JPL. Macos is primarily aimed at modelling the control of optical systems under external influences. It can perform geometric and diffractive propagation of wavefronts, but has no support for modelling atmospheric turbulence. Macos is written in Fortran and has a text-based user interface with pgplot plotting capabilities. I don't believe Macos has any parallelization support. Macos is considered proprietary, and may be commercialized in the future.

Beam Warrior is a commercial simulation developed in Europe for purposes similar to Macos. This simulation supports diffractive and geometric propagation, and has a novel approximation to diffractive propagation using gaussian beam decomposition. Beam warrior also has the capability of accurately tracking radiosity information in geometrical optics simulations. This simulation does not contain functionality for modelling atmospheric turbulence. I do not know what programming language this project uses, whether it supports parallelization, or the user interface that it provides.

GLAD is a commercial simulation written by a company called Applied Optics Research. This simulation has fairly sophisticated support for performing diffractive propagation through optical systems. I have a copy of the manual for this software, but almost nothing appears to be available over the web. I don't know anything else about this simulation - even the programming language it uses. There used to be some reports on the web, but they seem to have disappeared.

Light Pipes is a simulation that models diffractive free space propagation and the effects of various optics such as apertures and lenses. The original simulation used the pipe command to transfer the wavefront data between individual programs that effected the free space propagation and the optical transformations. The programs themselves were written in C. There is now a Matlab version of this software, though I'm unfamiliar with how this version operates. This simulation contains no support for modelling atmospheric turbulence, and no parallelization support. The source code is freely available on the web.

See is a Scidar simulation written by Miles Adcock and Chris Dainty. This simulation supports multipath diffractive propagation, though the problem geometry is somewhat limited. A manual used to be available on the web, but seems to have been removed. To my knowledge the source code was never available, and I don't know which programming language See uses.

Wave Train is a single conjugate AO simulation distributed by MZA Associates. This simulation relies on a low level C++ library culled from previous simulation efforts for the computationally intensive parts of the simulation. It also provides a sophisticated GUI, and I get the impression from the documentation that the GUI was one of the primary goals of the project. This simulation is capable of modelling atmospheric turbulence and contains a quite sophisticated model for a single conjugate AO system. I do not know whether this simulation supports diffractive propagation or parallelization. This company seems to be actively running simulations for the Airborne Laser project, so presumably work is continuing on this simulation. There is a fair amount of documentation for the GUI available on the web, but little information about the underlying C++ library. This simulation was developed at Los Alamos, and I believe that it is free to anyone working under government contract. The company does charge for support.

Among the simulations mentioned in the Wave Train documentation as forming the basis for their C++ library, I was able to find some web literature on TASAT. This simulation effort is aimed towards satellite imaging with AO. TASAT was written by Northrop-Grumman.

Over the past several years, ESO has been developing an AO simulation for use on eight meter telescopes. This simulation is capable of modelling laser beacons, including the uplink. It also contains support for both Shack-Hartmann and curvature sensors, for separate deformable mirror and tip tilt

correction, and models for reconstructors and control loops. This simulation does not contain support for diffractive propagation. It is written in IDL and includes a GUI. This project was funded under ESO's "Training and Mobility Researchers Network". While I'm not sure, I believe funding for this simulation may have ended. To my knowledge this software is not being publically released.

Francois Rigaut has another single conjugate AO simulation. I'm not sure whether this simulation is in any way related to the ESO one. It seems to share many of the same features. These include support for both Shack-Hartmann and curvature sensors. This software is written in IDL, and is freely available on the web. Apparently this project has a fairly large user community.

The Optical Sciences Company has its own single conjugate AO simulation called AOTools. This software is written in object oriented Matlab. There is documentation available on this company's website. The software is quite interesting in that it clearly contains algorithms that encapsulate some fairly sophisticated experiential knowledge of single conjugate AO systems possessed by this company. This software is proprietary, but I believe that it is possible to buy a copy.

Lawrence Livermore had developed a simulation used in the design of the Keck AO system. Don Gavel, Jose Milovich and Scott Wilkes are currently reimplementing this code in C++ and extending it to include functionality for MCAO simulations. The project specifically aims to support parallelization. I believe there is no publically available documentation for this project.

Rodolphe Conan is developing a single conjugate AO covariance simulation. This simulation is written in C++ and includes parallelization support. Some documentation may be available from the author.

John Breakwell and collaborators at Lockheed-Martin are currently developing an MCAO simulation. I believe the programming language is Matlab, and I'm uncertain as to whether they intend to support parallelization. This effort is proprietary, and to my knowledge no documentation is publically available yet.

Brent Ellerbroek has written a simulation of MCAO systems which has the most sophisticated capabilities for generating MCAO reconstructors available to date. This simulation includes support for Shack Hartmann wavefront sensors and deformable mirrors. I am uncertain as to whether this software can simulate diffractive propagation. This software is written in a combination of Ratfor and Matlab, and some parallelization capabilities were recently added. There is also some documentation of this simulation in the published literature (Ellerbroek & Cochran 2002);

Miska Le Louarn has also written an MCAO simulation. Originally written in IDL, this simulation has recently been reimplemented in C and parallelization support was added using MPI. This simulation is not capable of modelling the diffractive propagation of wavefronts. Some documentation of this simulation appears in the published literature (Le Louarn 2002).

In addition to the above simulations, there are a couple of other projects that have relevance to specific applications described in the previous section. Gary Chanan and Mitch Troy have developed software to perform segment phasing on the Keck telescope. They also have software to simulate PSF's from GSMT's. To my knowledge this software is not being publically released, and there is no documentation available. Walter Wild wrote a reconstructor generation package for single conjugate AO systems called A++. This software runs on Windows, and the executable is freely available on the web. Unfortunately the source code is not, and may have been lost. Finally, there is undoubtedly a large amount of AO related software that individual members of the AO community have developed for their private use. Generally speaking, academic researchers are often willing to share their software, but are not able to provide much support or documentation. It would be nice to incorporate some of this into a project that does offer more formalized support and documentation.

Arroyo

A notable feature of the applications mentioned above is the large overlap between functionality required to support their simulation. All of these applications require functionality to propagate wavefronts through optics, and almost all of them require functionality to simulate wave propagation through turbulence. This functionality is repeatedly reimplemented in the simulations described above. This situation probably arises for a number of reasons. First, many of the simulations described above are commercialized products, and one can't expect the developers of such source code to share their work. In contrast, scientific researchers typically set out to solve a particular problem in simulation starting with experience in a single programming language. They may perceive reusing other software written in a different language to entail too great a burden on their time and resources. Some simulation projects attempt to provide libraries of function calls in languages like IDL or Matlab. These projects can fall short of what is required for GSMT simulations because it is difficult to do parallel programming in these languages. Finally, many of these projects strongly emphasize the development of a user interface that will allow a typical astronomer to design and perform simulations without any programming knowledge. This requirement can seriously limit the breadth of applicability of the project, and I would strongly advise against taking this approach for the simulations of AO systems on GSMT's.

The commonality among applications provides an opportunity to develop software that can serve a general purpose. Such software would not aim to provide simulations for all possible applications. Instead, it would be viewed as a basis from which specific applications could be developed. This is a very common paradigm in software development. Typically functionality is implemented as a library, and other applications can link against this library to use its functionality. The C standard library and any number of Fourier transform libraries and linear algebra libraries provide examples. The benefit of this approach is that development time and resources may be leveraged over multiple applications, and support for the library becomes a community-wide priority. Broad input can serve to improve the quality of the library interface, and widespread use of this interface results in a much more thoroughly tested code base. Consequently, development time for projects utilizing this code is greatly reduced. This is a much stronger paradigm than the one that governs the simulations described above, in which typically a single researcher or company is attempting to develop and support a simulation project.

The Arroyo project currently under development at Caltech consists of a set of C++ libraries that aim to provide reusable functionality for the simulation of electromagnetic wave propagation through a turbulent atmosphere, telescope optics and a single or multiconjugate AO system. Such functionality may then be used by programmers to perform simulations of many of the applications mentioned above. Ultimately Arroyo aims to support Scidar, Lidar and GSM simulations, and simulations of single and multiconjugate AO systems. Arroyo's functionality will also be useful for calibration applications mentioned above. Arroyo will provide statistically valid data to downstream simulations of instrumentation and data analysis algorithms. These downstream simulations can then be written by those who develop the instrumentation or algorithms. At a future date Arroyo may be able to incorporate external influences on the telescope and AO optics that may be generated from other modelling programs. Currently feedback for active control of these external inputs is not contemplated. Finally Arroyo contains no facilities for testing implementations of real-time computing systems.

Arroyo aims to support extensibility of the library functionality through object oriented design. Arroyo does so by providing a library with an abstract programming interface (API) that includes a number of base classes. Classes derived from these base classes can reimplement functions in the API, and programs that rely on this API need not be modified as the class hierarchy is extended. Extending the library through inheritance may be achieved in a number of different ways. Users may write derived classes and

Figure 1: Illustration of multipath propagation through turbulence. This figure has not yet been manufactured...

link them into a program at compile time. They may choose to formalize these classes by incorporating them into another library, which they themselves may wish to distribute to other users. Finally, classes could be contributed back into Arroyo itself for inclusion in a future release.

Class Design Description

For its first public release, Arroyo aims to support functionality for geometric and diffractive propagation of wavefronts through turbulence. In subsequent releases, Arroyo will support propagation of these wavefronts through various types of AO systems. This section contains an overall description of the higher level features of Arroyo supported in the first release. This section does not attempt to describe every class in the project. Instead, it attempts to describe the higher level classes and their key member functions that collectively provide the functionality to run the simulation shown in Figure 1. After reading this section, it is my hope that a new user with some C++ experience may start to investigate the example programs included in the distribution. Full cross-linked HTML documentation for all classes and functions in the libraries is available on the Arroyo home page.

In the next few sections, I specifically identify base classes for certain hierarchies, and describe their key virtual member functions. These base classes provide an abstract representation of a certain concept, and the virtual member functions provide an abstract interface for this concept. For each of these base classes, I provide a description of one or more derived classes that I have created for Arroyo. Each of these base classes provides an opportunity to extend the library functionality through derivation of new classes. Implementation of this extension requires that a new realization of the virtual member functions to be provided for the new derived class. Thus, if your favorite model or concept is absent in the description below, you can extend Arroyo to include the missing functionality by providing an algorithmic implementation of the virtual member functions.

Coordinate Free Geometric Programming

Keeping track of the geometric details apparent in Figure 1 is critically important to the simulation. The typical approach in optics simulations is to specify all objects in the simulation relative to a single point, which is usually taken to be the center of the telescope aperture. In modelling the interaction between two objects in the simulation, I felt it was more sensible to work in coordinate systems local to the objects, rather than an arbitrary coordinate system. Therefore, one of the early choices I made was to use a coordinate free geometric programming approach (Siggraph 1989, De Rose 1989), which I implemented as a set of classes that represent geometrical concepts. Hopefully the utility of this choice will become more clear as this description proceeds.

The classes **three_point** and **three_vector** are used to specify points and vectors in a three dimensional space. The difference between a **three_point** and a **three_vector** is that the latter is invariant under translation. With this distinction, one can add a **three_point** and a **three_vector** to get another **three_point**, or subtract two **three_points** to get a **three_vector**. One can also take dot or cross products of two **three_vectors** - the former returns a scalar and the latter returns a **three_vector**.

The notion of a coordinate frame is encapsulated by the class **three_frame**. This class inherits the class **three_point**, which is interpreted as the origin of the frame. Therefore a **three_frame** may be passed directly to all functions taking a **three_point** as an argument, and all member functions of **three_point** may be invoked by a **three_frame**. Thus for instance, one may subtract from an arbitrary **three_point** a **three_frame** to get a vector that may be interpreted as the offset of the **three_point** from the origin of the **three_frame**.

The class **three_frame** has a set of member functions **three_frame::x()**, **three_frame::y()**, and **three_frame::z()**, which return a **three_vector** corresponding to the x, y and z axes respectively. One may retrieve the coordinate of a **three_point** in a particular **three_frame** through the member functions **three_point::x(const three_frame &)**, **three_point::y(const three_frame &)**, and **three_point::z(const three_frame &)**. Likewise, one may retrieve the component of a **three_vector** in a particular **three_frame** through the member functions **three_vector::x(const three_frame &)**, **three_vector::y(const three_frame &)**, and **three_vector::z(const three_frame &)**.

Construction of these geometrical objects may be carried out using the following constructors

```
three_point::three_point(double, double, double, const three_frame &)  
three_vector::three_vector(double, double, double, const three_frame &)  
three_frame::three_frame(const three_point &, const three_vector &,  
const three_vector &, const three_vector &)
```

The first two constructors specify the coordinates or components of the point or vector in the given three frame. The last constructor specifies the origin and axes of the coordinate frame. The axes must be orthogonal or this constructor throws an error.

A **three_transformation** is a base class used to represent an arbitrary geometrical transformation. There is a subset of such transformations that are orthonormal. These are represented by another base class **three_orthonormal_transformation**, which inherits **three_transformation**. Orthonormal transformations consist of an arbitrary composition of translations, rotations, and reflections. Each of these are represented by the classes **three_translation**, **three_rotation** and **three_reflection**, respectively. An example of a non-orthonormal transformation is given by the class **three_scaling**, which inherits **three_transformation** directly.

The two key virtual member functions of the base class **three_transformation** are

```
three_transformation::transform(three_point &)  
three_transformation::transform(three_vector &)
```

These functions affect a transformation of the corresponding object.

The class **three_orthonormal_transformation** contains the virtual member function

```
three_transformation::transform(three_frame &).
```

A transformation on a **three_frame** is interpreted as a passive transformation in Arroyo. For example, if one declares a **three_translation** and uses it to transform either a **three_point** or a **three_frame**, then the coordinates of the **three_point** in the **three_frame** are identical in the two cases.

Construction of these transformations may be carried out using the following constructors

```
three_translation::three_translation(const three_vector &)  
three_rotation::three_rotation(const three_point &, const three_vector &, double)  
three_reflection::three_reflection(const three_point &, const three_vector &)  
three_scaling::three_scaling(const three_point &, double)  
three_scaling::three_scaling(const three_point &, const three_vector &, double)
```

The first constructor is self-evident. The second declares a rotation at a particular point around a particular axis. The third argument is the magnitude of the rotation. The third constructor specifies a reflection across a particular plane, which is specified using a **three_point** and a **three_vector**. The fourth constructor specifies an isotropic scaling about a particular point. The fifth constructor specifies a scaling at a particular point along a particular axis. The scaling factors for these last two constructors are specified by their final argument.

The concepts of **three_point**, **three_vector** and **three_frame** are used repeatedly throughout Arroyo to specify the geometrical locations and orientations of objects in the simulation. For example, these geometrical concepts may be used to keep track of complex geometrical arrangements of natural and laser guide stars, and for tracking the pistons and tilts of each element on a segmented mirror telescope. Many classes directly inherit these classes, so that all these functions may be applied directly to the derived objects. Geometrical transformations of objects in the simulation are straightforward through the application of the above **three_transformation** classes.

Electromagnetic Wavefronts

In this section I describe the classes used to simulate the generation, free space propagation, optical transformations and detection of electromagnetic wavefronts.

Emitters

The class **emitter** serves as the base class for classes that serve as a source of electromagnetic wavefronts. Ultimately the class hierarchy will grow through derivation to include pointlike stars, extended sources like galaxies, and sources with three dimensional structure like LGS spots. These types of emitters will know everything about themselves that they need to generate a wavefront. This includes the radiosity and any temporal or spectral dependence of the source. The emission process will be described in more detail in the next section.

So far only the simplest two possible emitters have been implemented. A **plane_wave_emitter** is an emitter that represents wavefronts from a source distant enough that any curvature of the wavefront can be neglected. The class **plane_wave_emitter** inherits both **emitter** and **three_vector**. The latter class is interpreted as the direction of emission. Likewise a **spherical_wave_emitter** is a class that represents a pointlike source of wavefronts that is close enough that curvature of the wavefront may not be neglected. The class **spherical_wave_emitter** inherits both **emitter** and **three_point**. The latter class serves to specify the spatial location of the emitter. Note that the geometrical transformations described in the previous section may be used to rotate an instance of a **plane_wave_emitter** or translate an instance of a **spherical_wave_emitter**.

Construction of the above emitters may be carried out through the following two constructors.

```
plane_wave_emitter::plane_wave_emitter(const three_vector &)  
spherical_wave_emitter::spherical_wave_emitter(const three_point &)
```

Currently there is no support for radiosity or spectral information in these classes. In the future these will be added and more general constructors will be provided.

Wavefronts

Arroyo represents wavefronts as two dimensional complex arrays. These wavefronts really represent a finite piece of an infinite surface. The class used to represent wavefronts in Arroyo is the template class **diffractive_wavefront**<T>. A template class is a class whose realization is a function of another class (Stroustrup 1997). One can replace the T within the angle brackets with either another class or an intrinsic type to create a particular type of diffractive wavefront. In Arroyo you may choose to instantiate a diffractive wavefront using the intrinsic types float or double. This allows you to store wavefronts in single or double precision. The name **diffractive_wavefront**<T> was chosen in case ray tracing functionality is added to Arroyo at a future date.

Wavefronts have a number of properties. They have a wavelength, a pixel scale, a scalar curvature to represent the parabolic component of the phase and a 2d array to specify the pixel dimensions of the complex array. They also have a timestamp. All of this information is actually stored in another template class **diffractive_wavefront_header**<T>, which is inherited by **diffractive_wavefront**<T>. The benefit to having the ability to instantiate a class representing just the header information of a wavefront is that one can do quite a lot of useful things without having to carry around a large array of data. For example, suppose you have run a simulation and have generated 10,000 disk files each holding a single wavefront. If you want to perform some tests on the wavefronts or find a particular wavefront using a program, you can load only the kilobyte of header information by instantiating a **diffractive_wavefront_header**<T> rather than loading the potentially much larger quantity of data necessary to instantiate a **diffractive_wavefront**<T>. The class **diffractive_wavefront_header**<T> also inherits **three_frame**, which is interpreted as the geometrical location and orientation of the wavefront. In particular, the wavefront is interpreted as propagating along the z axis of the frame, with the pixel axes of the wavefront aligned with the x and y axes of the frame.

Instances of the class **diffractive_wavefront_header**<T> may be constructed with the following constructor.

```
diffractive_wavefront_header<T>::diffractive_wavefront_header(const vector<long> &,
                                                             const three_frame &,
                                                             double, double, double)
```

The first argument is a two dimensional **vector** containing the pixel dimensions of the wavefront. I note in passing that the class **vector** is part of the C++ Standard Template Library. The second provides the wavefront frame of reference. The last three arguments are the wavelength, pixel scale and curvature of the wavefront, respectively.

Free space propagation in Arroyo is affected through the following member functions of **diffractive_wavefront**<T>.

```
void diffractive_wavefront<T>::exact_propagator (double, double, vector<long> &)  
void diffractive_wavefront<T>::near_field_angular_propagator(double)  
void diffractive_wavefront<T>::near_field_fresnel_propagator(double)
```

```

void diffractive_wavefront<T>::far_field_fresnel_propagator(double)
void diffractive_wavefront<T>::far_field_fraunhofer_propagator(double)
void diffractive_wavefront<T>::
    far_field_fresnel_goertzel_reinsch_propagator(double, double, vector<long> &)
void diffractive_wavefront<T>::
    far_field_fraunhofer_goertzel_reinsch_propagator(double, double, vector<long> &)
void diffractive_wavefront<T>::geometric_propagator(double)

```

The first propagator performs the exact propagation, which is an N^4 process. The first argument is the propagation distance, the second is the pixel scale of the resulting wavefront, and the final argument is a 2d vector containing the pixel dimensions of the final wavefront. The second and third propagators perform near field propagation via a pair of 2d fast Fourier transforms using the angular and Fresnel approximations, respectively (Goodman 1993). Both take the propagation distance as the sole argument. The fourth and fifth propagators perform far field propagation via a 2d fast Fourier transform using the Fresnel and Fraunhofer approximations, respectively (Goodman 1993). Again, both take the propagation distance as the sole argument. The sixth and seventh propagators perform far field propagation via the Goertzel Reinsch algorithm using the Fresnel and Fraunhofer approximations, respectively (Papalexandris & Redding, 2000, Stoer & Burlisch 1993). This algorithm employs a clever recursion relationship for performing the 2d fast Fourier transform that allows one to choose arbitrary pixel scale and pixel dimensions for the final wavefront. For these propagators the first argument is the propagation distance, the second argument is the final pixel scale, and the third argument is a 2d vector containing the pixel dimensions of the final wavefront. The final propagator effects a geometric propagation by simply translating the origin of the wavefront's three frame and rescaling the pixel scale if the wavefront has nonzero curvature. All of these propagators update the timestamp in the header by an amount equal to the propagation distance divided by the speed of light.

One of the difficulties of applying fast Fourier transform based free space near field propagators is that in propagating a finite piece of the wavefront the edges of the array become corrupted (Johnston & Lane, 2000). To understand this effect, let us first consider how a wavefront of infinite extent propagates. This propagation may be viewed as a convolution of the wavefront with a propagation kernel. As a function of lateral displacement from a point on the initial wavefront, this kernel is a complex function with unit amplitude and parabolic phase. The coefficient on the parabola is inversely proportional to the product of the propagation distance and the wavelength. Contributions to a point on the final wavefront effectively terminate when the phase is wrapping much faster than any fluctuation in the initial wavefront, at which point all these contributions average away.

Now consider the situation where the infinite wavefront is represented using a finite piece. In this case a cyclic convolution is performed by Fourier transforming this piece of the wavefront, multiplying by the transform of the kernel, and transforming back. Because the wavefront is never truly periodic, the edge discontinuity introduces ringing in the final wavefront amplitude and phase in the vicinity of the edge. This ringing extends into the wavefront a distance determined by the coefficient of the parabola - roughly speaking it terminates when the parabolic phase runs through a full turn over a single pixel in the wavefront. To avoid the corrupted part of the wavefront one must propagate a larger piece of the wavefront than one is actually interested in using, and then discard the edges.

It would be possible to embed this choice directly into the API, but one may wish to try out different algorithms for the suppression of this ringing depending on the goals of the simulation. One is better off attempting to abstract away the details so as to keep the interface clean. To this end I've written a base class **propagation_plan**. The key virtual member function of **propagation_plan** is

```
diffractive_wavefront_header<T>  
propagation_plan::pad(const diffractive_wavefront_header<T> &, double)
```

The second argument is the distance over which the wavefront will be propagated. This function returns an appropriately padded diffractive wavefront header. This class is inherited by **geometric_propagation_plan** and **near_field_propagation_plan**. The former returns an unmodified copy of the wavefront header, while the latter pads the axes by a factor that depends on the pixel scale and wavelength of the wavefront and the distance of propagation. I have ideas for another propagation plan, but these will have to wait for the time being.

Lastly, given the notion of a wavefront we can describe the key virtual member function for the **emitter** class.

```
diffractive_wavefront<T> emitter::emit(const diffractive_wavefront_header<T> &)
```

This member function takes a wavefront header as an argument and returns a diffractive wavefront. It may use the wavelength and timestamp in the header if necessary to initialize the complex array with the correct values. Note that this member function is returning the piece of the wavefront specified by the pixel scale, axes and the **three_frame** of the **diffractive_wavefront_header<T>** passed as the argument to the function. Depending on the properties of the emitter, the resulting wavefront may have a finite curvature.

Detectors

For the simulations described above we would like to be able to represent a wide variety of detectors used for both science imaging and in wavefront sensors. As the sophistication of such simulations grows, we would like this representation to be able to increase in fidelity by taking into account more subtle detector effects. These would include models for infrared arrays and CCD's that incorporate quantum efficiency, read noise, dark current, bias, bad pixels, spectral responsivity, and bleeding. Naturally one would like to be able to specify the physical size of the pixels and the dimensions of the array.

Arroyo does not yet contain a class to represent a detector. However, I have a rough conception of the interface that such a class would require. Naturally we want a base class to represent all the possible derived classes for the different detectors. Let us call this base class **detector**. The key virtual member functions of such a class might be something like

```
void detector::detect(const diffractive_wavefront<T> &)  
void detector::detect(const diffractive_wavefront<T> &, const emitter &)  
observation detector::readout()
```

The first member function would act to detect a wavefront - typically by adding its squared amplitudes into an existing 2d array. The second member function would be used for emitters of finite extent. The detection would consist of convolving the emitter with a PSF formed from the wavefront. The final function would read out the detector, constructing and returning an as-yet-to-be-written observation class designed to hold optical and infrared observations. Classes derived from this base class would be able to reimplement these functions in whatever manner the class designer wished to employ. In particular note

that for single photon counting detectors the PSF formed from the wavefront could be interpreted as a probability distribution from which a photon is drawn by the detect member function above.

Optics

Optics are a hugely important part of Arroyo, and have quite a deep class hierarchy associated with them. There are actually four base classes for this hierarchy. One of the reasons I have so many base classes is that this particular hierarchy is the one most likely to be extended by other users. Because of this I tried to leave a lot of room for new development.

The base class for all optics in Arroyo is called **optic**. The key concept associated with this base class is a flag option to neglect or account for foreshortening of the optic. That is, it determines whether to include the effects of a wavefront striking an optic at an angle. For example, if a wavefront is incident at some angle on a circular aperture, then the flux that actually passes through the aperture is bounded by an elliptical region if the effect of foreshortening is included. Inclusion of this effect can often give rise to a fairly substantial computational penalty, and since many of the angles in the problem are very small, it is useful to be able to turn off this effect. In the example above, the wavefront would simply be assumed to be incident normally on the aperture for purposes of the transformation, so that the resulting transmitted wavefront would be bounded by a circular region.

The second base class in the optic hierarchy is **plane_optic**. This base class represents optics that may be approximated as a planar surface. Apertures are a good example of this type of optic. This base class inherits both **optic** and **three_frame**. Arroyo's convention is that the z axis of this frame is normal to the surface of the optic, while the transverse dimensions are interpreted differently depending on the derived class.

The third base class in the optic hierarchy is **one_to_one_optic**. This base class is meant to represent optics that have one input wavefront and one output wavefront. This class inherits the **optic** class. The key virtual member function for this base class is

```
void one_to_one_optic::transform(diffractive_wavefront<T> &) const
```

The final base class in the optic hierarchy is **one_to_many_optic**. This base class is meant to represent optics that have one input wavefront and multiple output wavefronts. The pyramidal element from a pyramid sensor would be an example of this type of class, as it splits one input wavefront into four output wavefronts. This class inherits the **optic** class. The key virtual member function for this base class is

```
vector<diffractive_wavefront<T> >  
one_to_many_optic::transform(const diffractive_wavefront<T> &) const
```

One could also imagine a many to one optic or a many to many optic, but so far I haven't found any application for these.

Having set up the class hierarchy with all of these different types of optics, I will now describe the types of optics that are currently supported by Arroyo. The first type of optic are apertures. Apertures are simply holes of various shapes in opaque screens. These are represented by the base class **aperture**, which inherits the classes **plane_optic** and **one_to_one_optic**. There is one flag option associated with apertures, which affects their transform member function. This flag dictates whether or not to weight wavefront pixels straddling the edge of the aperture by the area of the pixel that actually lies within the

aperture. This may seem like a fairly minor effect, but it actually turns out to be helpful in representing the very narrow gaps between the segments of GSMT's.

There are a number of different shapes of apertures, each of which is represented by a class derived from the **aperture** class. These include **circular_aperture**, **annular_aperture**, **rectangular_aperture** and **hexagonal_aperture**. In addition, there is a **spidered_annular_aperture** derived from **annular_aperture**. None of these classes are stored as two dimensional arrays of pixels. Instead, minimal geometric information about their dimensions is stored. Each of these classes inherits **three_frame** via **plane_optic**, and the origin of this **three_frame** is taken to specify the center of the aperture. For the class **rectangular_aperture**, Arroyo follows the convention that the edges of the aperture are aligned with the transverse axes of the **three_frame**. For the class **hexagonal_aperture** one vertex of the hexagon is taken to be along the y axis of the frame. Finally, the class **spidered_annular_aperture** can have an arbitrary number of spiders of arbitrary width, and one of the spiders is always taken to lie along the positive y axis of the frame.

The second type of optic currently supported by Arroyo is an infinitesimally thin phase changing screen that is used to approximate the atmospheric turbulence. In Arroyo the screen is represented by a class called **refractive_atmospheric_layer**, which represents the screen as a two dimensional real array of values representing optical path differences. In the next three sections I will describe the classes that supply the functionality used to construct instances of this class. Here I will confine the discussion to the information this class contains and how the transform member function is implemented.

Like the class **aperture**, the class **refractive_atmospheric_layer** inherits the classes **plane_optic** and **one_to_one_optic**. In addition to this functionality, this class possesses a pixel scale and a **three_vector** used to specify the wind direction. This class inherits **three_frame** via **plane_optic**, and the origin of this **three_frame** is taken to specify the geometrical center of the array. By convention the z axis of the frame is normal to the layer, and the transverse axes are aligned with the axes of the two dimensional array. It is important to emphasize that there is no convention that the wind vector must be aligned with one of the axes of the frame.

So far all instances of the class **refractive_atmospheric_layer** are represented in double precision. In the near future I would like to relax this restriction by promoting **refractive_atmospheric_layer** to a template class like **diffractive_wavefront<T>**.

One rather challenging aspect with the implementation of the transform member function for the **refractive_atmospheric_layer** class is in adding the pixellated layer to the pixellated wavefront. This was a problem that I avoided in the implementation of the aperture transformations because the apertures themselves are not pixellated. In that case it was relatively easy to determine whether the pixel lay within the geometric aperture or not. Solving the problem of adding two pixellated surfaces together is rather more difficult. Since I am actively improving the algorithms I'm using for this purpose, it would be premature to comment on them.

Simulation of Turbulence

Having completed a description of the emission, free space propagation, optical transformations and detection of wavefronts, I now proceed to a discussion of the classes used to generate an instance of the class **refractive_atmospheric_layer**. There are a web of relationships in this part of the problem, and the classes have correspondingly more complex member functions.

Power Spectra

There are a variety of power spectra that are commonly used to describe atmospheric turbulence (Sasiela 1994). The simplest is the Komolgorov power spectrum, which is simply a power law in spatial frequency of the form

$$\Phi(\kappa) = C_n^2 \kappa^{-11/3}$$

Here κ is the spatial frequency. This power spectrum may be modified to include the effects of an outer scale. The most common analytic representations are the von Karmann and Greenwood forms. The former is given by

$$\Phi(\kappa) = C_n^2 (\kappa^2 + \kappa_0^2)^{-11/6}$$

and the latter by

$$\Phi(\kappa) = C_n^2 (\kappa^2 + \kappa_0 \kappa)^{-11/6}$$

Here κ_0 is the spatial frequency corresponding to the outer scale. Likewise, an inner scale may be added to the power law. Analytic forms for the inner scale include exponential and Frehlich inner scales. These inner scales may be added to the power spectra above by including a multiplicative factor. For the exponential inner scale this factor is

$$\exp -(\kappa/\kappa_0)^2$$

Here κ_i is the spatial frequency corresponding to the inner scale. For the Frehlich inner scale, Arroyo uses the 4 term approximation to the Hill spectrum described in Sasiela (1994). Arroyo also has a class for representing step function inner scales. For atmospheric turbulence the actual form of the power spectrum can have a significant impact on AO system design and the resulting scientific observations, and we would like to be able to simulate fairly arbitrary spectra.

Arroyo contains a number of classes that permit representation of power spectra with arbitrary inner and outer scales, and arbitrary power law exponent and coefficient. Naturally Arroyo effects this representation through the use of base classes. Two such base classes are **inner_scale** and **power_law**. Both of these base classes have a virtual member function to retrieve a value at a particular frequency.

```
double inner_scale::value(double)  
double power_law::value(double)
```

where the argument is the spatial frequency. The class **inner_scale** serves as a base class for two derived classes **exponential_inner_scale** and **Frehlich_inner_scale**. The class **power_law** can represent a power law with arbitrary exponent and coefficient. It is in turn inherited by two derived classes **von_Karmann_power_law** and **Greenwood_power_law**. Some useful constructors for these classes are

```

step_function_inner_scale::step_function_inner_scale(double)
exponential_inner_scale::exponential_inner_scale(double)
Frehlich_inner_scale::Frehlich_inner_scale(double)
power_law::power_law(double, double)
von_Karman_power_law::von_Karman_power_law(double, double, double)
Greenwood_power_law::Greenwood_power_law(double, double, double)

```

The first two constructors take the desired inner scale as their argument. The next constructor takes the power law exponent and coefficient as arguments. The final two constructors take the power law exponent, coefficient and outer scale as arguments.

These classes serve to represent parts of the power spectrum, but one final class hierarchy is used to combine these classes into a single power spectrum. The base class for this hierarchy is **power_spectrum**, and the only derived class is **isotropic_power_law_spectrum**. This last class is obviously meant to represent isotropic power spectra. It contains an instance of a **power_law** and an optional instance of an **inner_scale**. The constructor is

```

isotropic_power_law_spectrum::isotropic_power_law_spectrum(const power_law *,
const inner_scale * = NULL)

```

Here the first argument is a pointer to the base class **power_law** and the second optional argument is a pointer to the base class **inner_scale**. If no second argument is supplied to this function, then the resulting **isotropic_power_law_spectrum** has no inner scale. The virtual functions **inner_scale::value(double)** and **double inner_scale::value(double)** are used in **isotropic_power_law_spectrum** so that additional types of inner scales and power spectra may be added in the future.

The primary use of the power spectrum is to provide a random realization of a two dimensional turbulent layer. This functionality is supported through the virtual member function

```

refractive_atmospheric_layer
power_spectrum::get_refractive_atmospheric_layer(const vector<long> &,
double,
const subharmonic_method &)

```

Here the first argument is a two dimensional vector containing the pixel dimensions of the layer, and the second argument specifies the pixel scale of the layer. The final argument is a base class that specifies a method to correct for a particular effect that arises in the generation of these screens. Both the problem and the class hierarchy headed by the base class **subharmonic_method** are described in the next section.

Subharmonic Methods

Instances of the class **refractive_atmospheric_layer** are generated in a very specific way. A two dimensional complex array is allocated, and elements of this array are filled in with complex values having amplitudes given by the **power_spectrum** and random phases. This array is then fourier transformed to yield another complex array. The real and imaginary parts of this array represent independent, random realizations of a turbulent layer with the statistics of the power spectrum used in their generation.

The difficulty with this approach is that the Fourier transform enforces periodicity of the resulting array. In contrast, for power spectra typical of atmospheric turbulence most of the power is in long spatial wavelengths. To accurately represent this power, we would need to be able to represent layers that are not periodic at all, but instead have large discontinuities across the edges. Several researchers have described techniques to modify the frequency space array so as to avoid generating turbulent layers with periodic boundary conditions (Lane, Glindemann & Dainty 1992, Johansson & Gavel 1994.) These techniques come down to adding power to the frequency space array so as to mimic the presence of spatial wavelengths longer than the physical dimensions of the turbulent layer. Hence these techniques are usually called subharmonic methods.

Different techniques have been proposed to effect this correction, and comparison of these techniques is one of the requirements for Arroyo. To this end, **subharmonic_method** is used as a base class for a hierarchy of potential techniques. The key virtual member function for this base class is

```
void subharmonic_method::apply_subharmonic_correction(const power_spectrum &,
                                                    const vector<long> &,
                                                    double, bool, double *)
```

Here the first argument is the power spectrum to use in the correction, the second argument is the pixel dimensions of the array, the third argument is the pixel scale, and the fifth argument is a pointer to the raw two dimensional complex array. The fourth argument actually specifies how the correction should be applied. I will describe this option in more detail in the section below. This declaration is a bit crude and unrefined since we have to pass a pointer to the raw data. However, it has only been used within the member functions of the **power_spectrum** class. I may look towards hiding this member function in the future. Arroyo users shouldn't really be calling this member function directly - I've described it here just to show how the functionality is implemented.

Arroyo contains three types of subharmonic methods, represented by three different derived classes: **null_subharmonic_method**, **Lane_subharmonic_method**, and **quad_pixel_subharmonic_method**. The first indicates no correction should be made. The second implements the algorithm of Lane, Glindemann & Dainty (1992). The third is one that I invented, and which I think works better than the **Lane_subharmonic_method**. The constructors for these classes are

```
null_subharmonic_method::null_subharmonic_method()
Lane_subharmonic_method::Lane_subharmonic_method(long)
quad_pixel_subharmonic_method::quad_pixel_subharmonic_method(long)
```

The second and third constructors take a number of subharmonic levels as an argument. Roughly speaking, this argument represents how many subharmonic frequencies should be introduced in the subharmonic correction.

Structure Functions

The final class related to turbulent screen generation is used to represent structure functions. This class is naturally called **structure_function**, and can represent both one and two dimensional structure functions. There is a one-to-one analytic relationship between power spectra and structure functions. It is

difficult to represent this relationship in software for the arbitrary case, so Arroyo supports the more restrictive functionality to generate a structure function from a power spectrum. This direction was chosen because Arroyo represents power spectra analytically and structure functions as arrays of sampled values. There are two virtual member functions of the class **power_spectrum** that return structure functions.

structure_function

power_spectrum::get_theoretical_structure_function(const vector<long> &, double)

structure_function

power_spectrum::get_expected_structure_function(const vector<long> &, double, const subharmonic_method &)

If the structure function corresponding to the power spectrum is known theoretically, the first function returns an instance. The first argument is a one or two dimensional vector corresponding to the pixel dimensions of the structure function. The second argument is the pixel scale. If the form of the structure function is not known theoretically, this function throws an error.

The second member function actually carries out the steps used in constructing a turbulent layer, but does not add the random phasor in frequency space. The result is a direct calculation of the structure function from the power spectrum (Johansson & Gavel, 1994). This function uses the **subharmonic_method::apply_subharmonic_correction** member function described above to analytically implement the subharmonic method without randomizing the correction. The fourth argument in this member function is used to specify whether the subharmonic correction should be applied randomly or not.

It is also possible to create a structure function by accumulating statistics from one or more instances of the class **refractive_atmospheric_layer**. One simply averages the square of the phase difference between points in the layer as a function of the spatial separation of the points. The class **structure_function** has a member function for this purpose.

void structure_function::add_statistics(const refractive_atmospheric_layer &)

This member function may be called repeatedly using layers generated from the member function **power_spectrum::get_refractive_atmospheric_layer** described above. In this way, a statistical estimate for the structure function can be created.

Figure 2 attempts to summarize the relationships between power spectra, structure functions, subharmonic methods, and refractive atmospheric layers described above. Arrows indicate methods to construct one class from another. For example, a power spectrum may be used to construct a structure function either theoretically or, together with a subharmonic method, by direct transformation. Likewise a refractive atmospheric layer may be used to construct a structure function in a statistical manner.

Other Simulation Classes

The classes described in the previous sections are sufficient to formulate a simulation that incorporates plane or spherical emitters generating wavefronts that propagate through turbulent atmospheres represented by multiple layers, through several types of apertures, and into the far field. However, to set up an arbitrary simulation of this nature is somewhat challenging. You would need to compute the size of the layers required to transform wavefronts from all emitters throughout the entire simulation. You would also need to specify the geometrical relationships for the layers and emitters. This would be a tedious

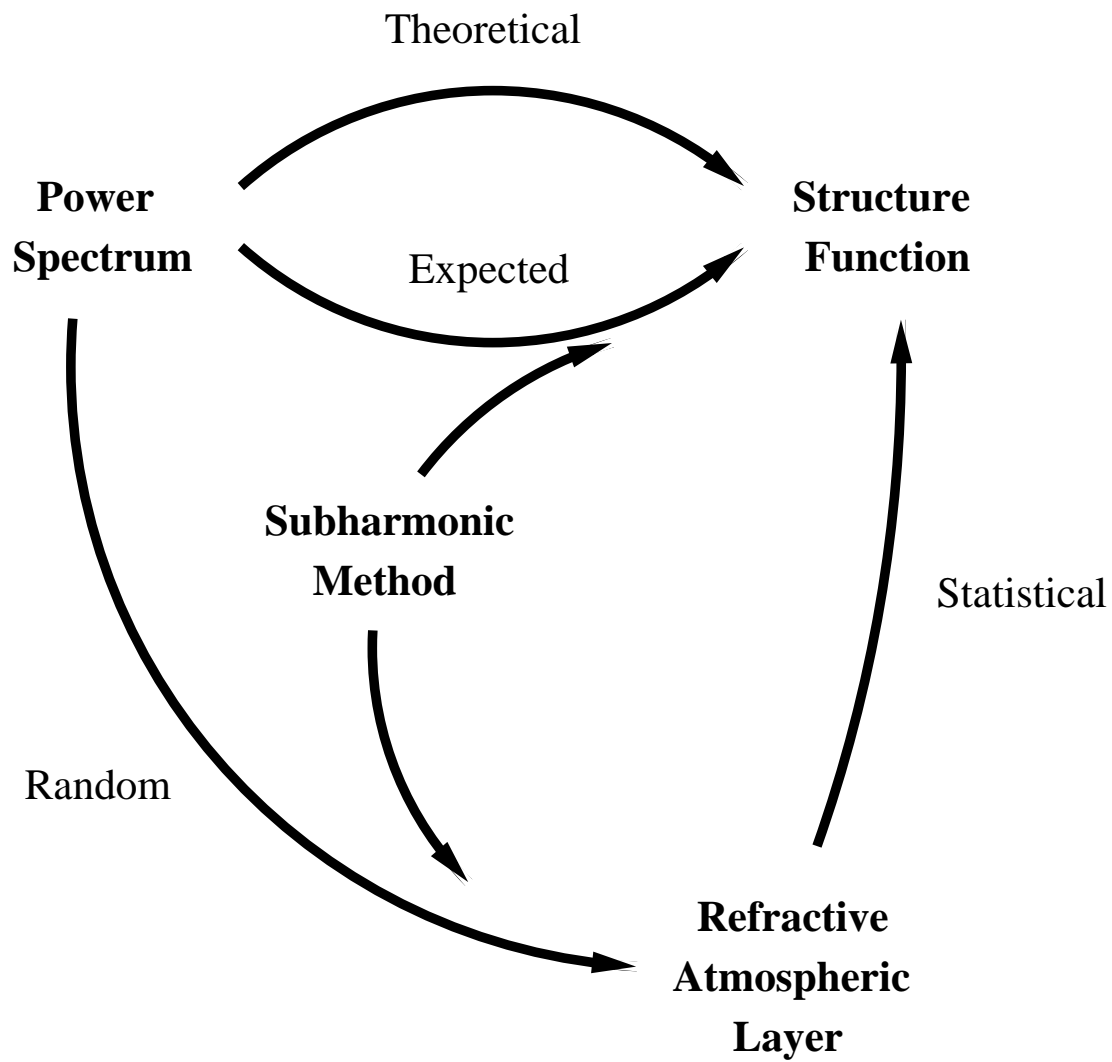


Figure 2: Class functionality for modelling turbulence

and error-prone process. To alleviate much of the setup required for running these simulations, I have written several additional classes that encapsulate the required functionality. In this section I describe these classes.

Vertical Turbulence Profiles

One useful concept worth encapsulating is the notion of a vertical turbulence profile. The class **Cn2_distribution** serves as a base class for a hierarchy of such classes. This class contains two useful virtual member functions.

```
double get_moment(double)
double get_integrated_profile(double, double)
```

The first virtual member function returns an integrated turbulence moment. It takes the exponent to use in the integration as an argument. For example, calling this function with zero as the argument will give the integrated C_n^2 profile, and calling it with 5/3rds as the argument will yield the turbulent moment related to anisoplanatism. The second virtual member function returns the C_n^2 profile integrated between two heights. The two arguments are the minimum and maximum height.

The class **Cn2_distribution** is inherited by the classes **Hardy_Cn2_distribution**, **SLCSAT_day_Cn2_distribution**, and **SLCSAT_night_Cn2_distribution**. The first class implements the turbulence model in equation 3.19 of Hardy (1998). The second and third classes implement SLCSAT models from equations 4.12 and 4.13 of Sasiela (1994). One can construct instances of these classes using the constructors

```
Hardy_Cn2_distribution::Hardy_Cn2_distribution(double, double,
                                              double, double,
                                              double, double,
                                              vector<vector<double>> > =
                                              vector<vector<double>> >(0))
```

```
SLCSAT_day_Cn2_distribution::SLCSAT_day_Cn2_distribution()
SLCSAT_night_Cn2_distribution::SLCSAT_night_Cn2_distribution()
```

The first constructor takes 6 arguments, plus one optional one, which are used to specify the strength and location of turbulence components. The first two represent the boundary layer coefficient and scale height, and the next two represent the tropospheric layer coefficient and scale height. The next three represent the coefficient, height, and width of the tropopause component. The final argument may be used to specify additional layers. This argument consists of a vector of vectors, in which the outer index specifies the layer and the inner index is a 3 component vector containing the coefficient, height and thickness of the turbulent layer. The other two constructors take no arguments.

Wind Models

It is useful to have a class hierarchy to encapsulate atmospheric wind models. This hierarchy is headed by the base class **wind_model**, which contains the virtual member function

```
vector<three_vector>  
wind_model::get_random_wind_vectors(const vector<double> &, const three_frame &)
```

This function returns a vector of instances of the class **three_vector**, which are meant to be used to specify the wind vectors of a set of turbulent layers. The function takes as arguments a vector of doubles interpreted as the heights of the layers and an instance of a **three_frame** whose origin is taken to specify the ground. The heights are measured along the positive z axis of this frame.

This class is inherited by the class **Hardy_wind_model**, which implements the wind model described in equation 3.20 of Hardy (1998). This wind model contains a constant ground layer component and a tropospheric component that is modulated vertically by a gaussian. The constructor for this class is

```
Hardy_wind_model::Hardy_wind_model(double, double = 0, double = 0, double = 0)
```

The first argument is the rms ground layer wind speed. The remaining three optional arguments specify the height and thickness of the tropopause wind component, and the rms tropopause velocity. Naturally this class reimplements the virtual member function **wind_model::get_random_wind_vectors**, and for concreteness I will describe exactly how this function is overridden. A random wind vector is chosen for each of the ground and tropospheric components. These vectors are drawn from two dimensional gaussian distributions which have an rms specified by the ground and tropospheric wind speeds. For each layer height, the ground wind vector is added to the tropospheric wind vector weighted by the vertical gaussian. The function returns the resulting set of wind vectors as a vector of instances of the class **three_vector**.

Refractive Atmospheric Models

The final class that I will discuss, **refractive_atmospheric_model**, aims to bring together all the pieces of the simulation. This class is somewhat restrictive, and really is still evolving. However it provides enough functionality to set up a simulation with a minimal amount of effort.

The class **refractive_atmospheric_model** contains an array of power spectra at different heights, a wind model, and a reference frame. First, an instance of this class may be constructed using

```
refractive_atmospheric_model::  
refractive_atmospheric_model(const vector<power_spectrum *> &,  
                             const vector<double> &,  
                             const three_frame &,  
                             const wind_model &)
```

The first and second arguments are vectors containing power spectra and heights, respectively. The third argument is a **three_frame** that is interpreted as having its origin at the ground. The heights of the power spectra are measured along the positive z axis of this frame. The final argument is the wind model.

This class should be used in two different stages. In the first stage, one can retrieve a **diffractive_wavefront_header<T>** instance for use with a particular emitter by specifying information about the aperture and propagation technique. The basic idea is that this function will figure out how large the wavefront from a particular emitter should be so that when you propagate it to the ground it will fully cover the aperture. It will also position the wavefront header at the altitude of the highest layer with transverse location appropriate for this emitter. The member function that provides this functionality is

```

diffractive_wavefront_header<T>
refractive_atmospheric_model::get_diffractive_wavefront_header(double, double,
                                                                emitter *, aperture *,
                                                                bool, propagation_plan *)

```

The first and second arguments are the electromagnetic wavelength and pixel scale of the wavefront. The third argument is a pointer to the emitter from which the wavefront will originate. The fourth argument is a pointer to the aperture that the wavefront must cover. The fifth argument indicates whether foreshortening of the layers should be enforced. The final argument is a pointer to a propagation plan.

Having run through all emitters and all electromagnetic wavelengths in the simulation using the above member function, one can use another member function of this class to retrieve a set of instances of the class **refractive_atmospheric_layer** that are sufficiently large to carry out the simulation. This function is

```

void refractive_atmospheric_model::
get_refractive_atmospheric_layers(const vector<double> &,
                                   const subharmonic_method &,
                                   const vector<diffractive_wavefront_header<T> > &,
                                   double, bool, bool,
                                   vector<refractive_atmospheric_layer> &)

```

The first argument to this function is a vector containing the pixel scales to use for each layer. The second argument is the subharmonic method to use in constructing the layers. The third argument is a vector containing all of the wavefront headers that will be propagated in the simulation. The fourth argument specifies the duration of the simulation. The fifth argument is a bool that specifies whether the pixel axes of each layer should be aligned with the wind vector of that layer, or whether they should all be aligned on a common set of axes. The sixth argument specifies whether foreshortening of the layer should be enforced. The final argument is a vector containing the instances of the class **refractive_atmospheric_layer** created by this function.

Having derived the wavefront headers and turbulent layers, it is a simple matter to call the **emitter::emit** member function, propagate the resulting wavefronts through free space to each layer in turn, applying the **refractive_atmospheric_layer::transform** member function. When the wavefront reaches the aperture, one calls the **aperture::transform** member function and then optionally propagates the wavefront to the far field.

The class **refractive_atmospheric_model** serves as a base class for two additional models that I have implemented - the **Ellerbroek_Cerro_Pachon_model** and the **Ellerbroek_Mauna_Kea_model** described by Ellerbroek & Rigaut (2000). The first is a six layer model and the second is an eleven layer model. Though the paper doesn't specify the power spectrum used, I assumed all layers had a Komolgorov power spectrum. These models are simply specific instantiations of **refractive_atmospheric_model**, and have no additional functionality. They have correspondingly simpler constructors.

```

Ellerbroek_Cerro_Pachon_model::Ellerbroek_Cerro_Pachon_model(const wind_model &,
                                                                const three_frame &, double, double)
Ellerbroek_Mauna_Kea_model::Ellerbroek_Mauna_Kea_model(const wind_model &,
                                                                const three_frame &, double, double)

```

The final two arguments in these constructors may be used to specify a Fried parameter at a particular wavelength. These arguments default to the nominal values stated by Ellerbroek & Rigaut (2000). For the Cerro Pachon model the default is 16.6 cm at .5 microns, and for the Mauna Kea model the default is 23 cm at .5 microns.

Finally, there is a simplifying assumption implicit in the **refractive_atmospheric_model** class that must be relaxed before the most general simulations may be performed. The emitters used to specify the geometry in the above member functions are all assumed to be above the highest turbulent layer. This precludes the simulation of the LGS uplink. It also is a problem for simulating Rayleigh beacons that are at altitudes lower than that of the highest layer. This assumption will be eliminated before the first public release of Arroyo.

Additional Design Details

This section attempts to describe some of the global conventions used in Arroyo.

Factories

A subtlety in the use of object oriented programming is that one would like to be able to construct an instance of a derived class and store a pointer to this instance without actually ever knowing which type of class was returned. The classic example is constructing an instance of a class from a file. In the object oriented paradigm we are only going to be calling virtual member functions on the instance, and so we don't want to have to know what type of derived class we've actually instantiated.

The classic solution to this problem is through the use of a function called a factory. The factory takes a certain number of arguments and returns a base class pointer to a dynamically allocated instance of the derived class. There are many ways to formulate this paradigm, but I've chosen a particular formulation and have attempted to stick with it throughout the project. The convention is that I declare these factory functions in the namespace of the base class. The name of the function is always the base class name followed by "_factory". There are typically two such factories for each base class: one for constructing an object from a file and one for constructing from an iofits object. Here are the prototypes for the factories in the **emitter** class.

```
static emitter * emitter::emitter_factory(const char * filename)  
static emitter * emitter::emitter_factory(const iofits & iof)
```

The uniform naming convention is designed to remind you to delete the pointer when you are finished with it. Otherwise you will get a memory leak.

The most straightforward way to write a factory for, say, constructing an object from a file is to open the file and look for some keyword in the header unique to a derived class. Based on the value of the keyword, one can simply dynamically allocate an instance using the derived class constructor and return the resulting pointer, which gets upcast to the base class pointer. The problem with this approach is that the definition of the factory is in the Arroyo library code, so that derivation of a new class from this base class requires modification of the Arroyo itself in order to use the factory. This runs contrary to Arroyo's stated extensibility goal that others should be able to write derived classes for their own use and then link these classes with Arroyo so that their executables can use the new classes with Arroyo's API.

The object factory paradigm (Alexandrescu 2001) provides a solution to this problem. This paradigm is extremely general, and to provide a concrete example I'll describe the specific choices used for this project. This description is intended for people who would like to write additional derived classes for Arroyo.

In the object factory paradigm every class is derived from a common base class. In Arroyo this class is called **AO_sim_base**. Each derived class must register a unique signature with the base class, along with a function pointer to the appropriate constructor for the derived class. The signature I chose was an STL map of strings to strings. This is a useful choice for classes stored as fits files, whose headers consist of key and value pairs. Each derived class in Arroyo has a unique value for the fits header keyword "TYPE". For example, the class **plane_wave_emitter** possesses the value "plane wave emitter". However, the paradigm can be used for signatures involving an arbitrary number of fits keyval pairs. When a factory function like the one above is called, it in turn calls a function that looks through the header of the fits file searching all the keyval pair maps that have been registered. When it finds a match, it calls the corresponding function pointer - the constructor for the derived class - storing the result as an **AO_sim_base** pointer. This pointer is then kicked back to the factory above, where it is downcast to the original base class using a dynamic cast.

The registration procedure that constructs the signature and passes it along to the base class together with the function pointer to the constructor is carried out in the source file containing the definition of the derived class. (Actually, all this registration is defined in an anonymous namespace so that the details don't leak into the API.) The functionality that performs the signature comparisons doesn't require any specific knowledge of the derived classes. Because of this, one may declare new classes and register them without modifying any of Arroyo's source code. This is exactly the behavior we wanted.

Error Handling

Arroyo uses the throw-catch feature of the C++ language (Stroustrup 1997). When an error condition is encountered within a routine during the operation of a program, that routine throws an error. The program then unwinds the stack to see if the function that called the routine will catch the error, meaning that there is functionality to handle the existence of the error. If the error is handled successfully, then program execution continues. If the error is not handled and the stack is unwound all the way up to main, the program exits with an error. While it is possible to throw an arbitrary class, all errors thrown out of Arroyo are strings. This isn't particularly good practice as it doesn't allow the routine that catches the error to take different actions depending on the type of error. Depending on how people end up using the library, I may attempt to improve on this situation.

There are typically three types of errors that can get thrown out of an Arroyo library call. First, a programmer can specify inconsistent arguments to a library function. Wave optics simulations are complex, and there are numerous opportunities for a programmer to inadvertently invoke Arroyo's library functions incorrectly. Arroyo purposefully makes little effort to attempt to determine what you actually meant when the arguments you have passed are incorrect or inconsistent. In cases like these the library will just throw an error with a message as to where the error occurred. The second type of error occurs when the program runs out of resources - typically RAM. This can occur when you attempt to allocate a Gigabyte array on a machine that has 128 Megabytes. Arroyo prints a message to this effect and throws an error. Finally, I wouldn't claim that Arroyo is infallible, and it is certainly possible that an internal error condition can occur. In this case Arroyo hopefully has a sufficient amount of internal error checking to detect that a problem exists, in which case an error is thrown. These are the situations you should report to me when you encounter them.

Random Seeds

The numerical simulation of turbulence - an inherently random process - requires the use of a pseudo-random number generator. Currently Arroyo simply uses the C library function `random()`. Control of the seed via the C library function `srandom()` is left to the programmer. In the example programs provided in the Arroyo distribution, specification of the seed is left as a flag option, and if uninitialized the random number generator is seeded with the output of the C library function `time()`. One can rerun the exact same simulation by seeding the random number generator with the same number. However, some care must be taken in attempting to run the same simulation with different input parameters. For example, if one attempts to rerun a simulation using a different electromagnetic wavelength and one chooses to use a near field diffractive propagator, the refractive atmospheric layers will be a different size because the wavefront requires a different amount of padding. If this padding is smaller, for instance, then a subset of the same random numbers used in the construction of the original layer will be used to construct the smaller layer, and the random turbulent realization will be different. A simple solution is to pass wavefront headers to the `refractive_atmospheric_model::get_refractive_atmospheric_layers` member function for all the wavelengths that you ultimately would like to simulate, even if you only perform the propagation at a subset of these wavelengths. You can always write these layers to a file and read them at a later time to perform propagation at another wavelength. Please keep this subtlety in mind if you are relying on repeating simulations. Arroyo also uses the random number generator in creating the random wind vectors from a wind model.

Namespaces

Namespaces are a C++ language feature that allows a programmer to group related functionality, and keep symbols like functions and variables out of the global scope (Stroustrup 1997). As a specific example, if a function `f()` is declared in a namespace `nmspc`, then a programmer can access this function through the syntax `nmspc::f()`. The use of namespaces in library design can help to avoid potential name clashes between symbols in different libraries. Classes are a type of namespace in which one may use the class name to resolve a member function of the class, and this is the reason all member functions described above have the class name prepended to them. However, namespaces are more general in that they may contain normal functions and variables that are not members of a class.

When you are writing a program, you cannot use a class or a function that is in a namespace without telling the compiler which namespace you are using. There are three ways to do this. You can prepend the namespace to the function, as above. You can tell the compiler that you will be using a particular function by placing a using declaration at the top of the program. In the example above, putting

```
using nmspc::f
```

at the top of the file would tell the compiler that every instance of `f()` found in the source code really means `nmspc::f()`. Finally, you place a using directive at the top of the program.

```
using nmspc
```

This tells the compiler that every symbol in the namespace `nmspc` should be placed in the global scope.

Arroyo wraps all simulation functionality into two namespaces: `AObase` and `AOsim`. All classes and functions that exist in the `AO_utils` library are placed in the `AObase` namespace. Likewise, all classes

and functions that exist in the AO_simulation library are placed in the **AOsim** namespace. For most purposes a programmer can place the using directives at the top of their source code and forget about these namespaces. However, provision of namespaces in a C++ library is just common courtesy.

File Format

Arroyo uses the fits file format. This has the advantage that wavefronts and images generated by Arroyo may be inspected and analyzed using standard astronomical tools like ds9 and IRAF. The multiple HDU facilities of the fits file format are used to store aggregate classes. The fits I/O library Cfitsio manages endian issues, thus ensuring that the files are portable to big or little endian machines.

Verification

There are two aspects to verification in Arroyo. The most straightforward is the regression testing included in the distribution. This consists of a collection of test programs that instantiate all the classes and test the member functions. Running make check in the Arroyo installation directory will execute all these programs, ensuring that they do not fail. Make check also generates a bunch of test files containing class instances, and you may find it useful to look through these and see how they were generated by the test programs. Regression testing is mainly used to catch errors that may occur when the software is installed on a new machine. This is particularly useful if you are attempting to port Arroyo to a different platform.

The second and more subtle verification step attempts to establish the consistency of simulations with analytic results. This is accomplished by repeatedly generating different refractive atmospheric layers, running simulations, and gathering statistical data from these simulations. For example, the two dimensional refractive atmospheric layers generated from a Komolgorov turbulence spectrum are known to have a structure function that grows with separation as $r^{5/3}$. By generating many such layers and forming the structure function directly, one may verify that this law holds to some level of fidelity. This test then verifies that the part of the code which generates refractive atmospheric layers from the Komolgorov power spectrm is returning statistically valid data. Other tests may be performed to verify the statistical properties of wavefronts after geometric or diffractive propagation through atmospheres with a given power spectral density and turbulence profile. Table 2 shows many such tests. Sasiela (1994) derives analytic results for a large number of cases using a unified theoretical approach. Analytic results for most of the tests listed in Table 1 are derived in this reference. When this is the case I have listed the section containing the derivation in the final column.

Memory Requirements

For realistic simulations involving multilayer atmospheres and large aperture telescopes, the memory requirements for storing the **refractive_atmospheric_layer** instances can rapidly exceed the maximum RAM available on a single workstation. Note that the maximum amount of RAM that may be added to a workstation is set by the number of available memory addresses. For a 32 bit processor this happens to be 2 Gigabytes. Though I expect the first applications of Arroyo will involve smaller, more manageable simulations, it is worth discussing some ways that this difficulty may be addressed in the future.

Prior to the start of the simulation one must generate a layer large enough to cover a metapupil formed from the telescope aperture and all of the emitters, for the entire duration of the simulation. This layer can easily have dimensions of order thousands of pixels on a side. One possible solution would be to generate each layer in turn and write it to disk. Then one could load from disk enough data from each layer to perform the simulation for a number of timesteps. One could then repeat this procedure for the next time interval, proceeding until the simulation was complete. The functionality required for this mode of operation would not be very difficult to supply in a future release.

This problem may also be addressed by moving to a 64 bit processor. Workstations containing such processors are currently available - the most economical ones are based on the Intel Itanium chip. One can currently buy a dual processor Itanium system with 4 Gigabytes of RAM for about \$5,000. Servers with up to 4 processors and 48 Gigabytes of RAM are available at substantial cost. I believe that as our simulations grow in sophistication together with our RAM requirements, the prices of these systems will drop to the point where we can think about using them.

Example Programs in the Distribution

In order to help users get started doing some simple simulations, a number of example programs are included in the Arroyo distribution. This section briefly summarizes the functionality of these programs.

The program **simhdr** may be used to print out the fits header of any file created by Arroyo. Multiple headers may be printed by specifying multiple filenames on the command line. The headers are output to stdout, so this output may be manipulated using Unix programs like grep and awk.

Test	Aperture	Power Spectrum	Propagator	Beam Shape	Reference
Zernike decomposition	Circular	Komolgorov	geometric	collimated	Noll (1976)
Zernike decomposition	Circular	von Karmann	geometric	collimated	Winker (1991)
Gradient tilt variance	Circular	Komolgorov	geometric	collimated	Sasiela §4.3
Zernike tilt variance	Circular	Greenwood	geometric	collimated	Sasiela §7.2
Zernike tilt variance	Circular	Komolgorov with exponential inner scale	geometric	collimated	Sasiela §7.3
Zernike tilt variance	Annular	Komolgorov	geometric	collimated	Sasiela §7.4
Gradient tilt variance	Annular	Komolgorov	geometric	collimated	Sasiela §7.4
Zernike tilt variance	Circular	Komolgorov	diffractive	collimated	Sasiela §7.5
Zernike tilt variance	Circular	von Karmann with exponential inner scale	geometric	collimated	Sasiela §12.5
Servo bandwidth	-	Komolgorov	geometric	collimated	Sasiela §4.8
Tilt anisoplanatism	Circular	Komolgorov	geometric	collimated	Sasiela §7.6
Tilt anisoplanatism	Circular	von Karmann	geometric	collimated	Sasiela §12.4
Power spectrum of tilt	Circular	Komolgorov	geometric	collimated	Sasiela §7.7
Power spectrum of tilt	Circular	von Karmann	geometric	collimated	Sasiela §12.6
Focal anisoplanatism	Circular	Komolgorov	geometric	collimated and focused	Sasiela §7.12
Focal anisoplanatism for extended sources	Circular	Komolgorov	geometric	collimated and focused	Sasiela §7.13
Focal anisoplanatism for offset sources	Circular	Komolgorov	geometric	collimated and focused	Sasiela §7.14
Scintillation	-	Komolgorov	diffractive	collimated, focused	Sasiela §4.7
Scintillation	Circular	Komolgorov	diffractive	collimated, focused	Sasiela §7.8
Scintillation from extended sources	Circular	Komolgorov	diffractive	collimated, focused	Sasiela §7.8
Scintillation	Circular	Komolgorov with exponential inner scale	diffractive	collimated, focused	Sasiela §7.8
Scintillation	Circular	Komolgorov with Frehlich inner scale	diffractive	collimated, focused	Sasiela §7.8
Scintillation anisoplanatism	-	Komolgorov	diffractive	collimated	Sasiela §7.10

Table 2: Analytic predictions for statistical averages. These predictions may be tested against those derived from simulations of wave propagation through turbulence.

The program **simple_simulation** performs a simulation with a single refractive atmospheric layer at the ground and a circular aperture. The user may specify the Fried parameter and velocity of the layer and the number of subharmonics to use in generating the layer. (The subharmonic method is always taken to be the **quad_pixel** one.) The user may also specify the size of the aperture, the timestep and duration of the simulation, the electromagnetic wavelength, the pixel scales of the layer and the wavefront, and the random seed. For each timestep in the simulation, the pupil plane wavefront is written to disk. The user may optionally propagate the wavefront to the far field and write this wavefront out instead.

The program **scintillation_simulation** constructs a single **refractive_atmospheric_layer**, applies this layer to a wavefront, and then performs a near field diffractive propagation of this wavefront over a series of steps in propagation distance. Most of the options available for the specification of the wavefront and the layer that are available in **simple_simulation** are also available in **scintillation_simulation**, with the exception that the simulation is carried out in propagation distance rather than in time. Like the program above, this program writes out the resulting wavefronts to disk.

The program **model_simulation** performs propagation of wavefronts from a grid of instances of the class **plane_wave_emitter** through an **Ellerbroek_Cerro_Pachon_model** or an **Ellerbroek_Mauna_Kea_model**. The user may specify the dimensions and field size of the grid, the pixel scales of the wavefront and the layers, and the tropospheric and ground layer properties of the **Hardy_wind_model** used in the formation of the **refractive_atmospheric_layer** instances. The user may choose to simulate geometric or diffractive near field propagation, and may specify the diameter of the circular aperture, the subharmonic depth, the random seed, and the timestep and duration of the simulation. The user may choose to apply one of two idealized corrections to the wavefronts: subtraction of the wavefront phase of the central emitter or subtraction of the average wavefront phase of all the emitters. Finally, the user may choose to propagate the wavefront to the far field. The program then writes out all of these wavefronts to disk.

The program **dwf2ppm** is a utility program I wrote to encode the wavefront amplitudes or phases in a portable pixmap (ppm) file. This graphics format is easy to generate, and the resulting files can be manipulated using any number of programs (e.g. ImageMagick's `convert` and `mogrify`). I use these files to generate mpegs of the simulations that I run, many of which are posted on Arroyo's home page. The program **dwf2ppm** will take as one of its arguments a colormap generated by ds9. (At the bottom of ds9's Color menu there is a "Save Colormap" menu item.) I wrote a little class to hold this colormap called **sao_colormap** that has some useful member functions for generating these ppm files. There is also a base class called **colormap** in case another colormap specification comes along. In this way it is easy to encode your ppm files in your favorite ds9 colormap.

Usage information for each of these programs may be viewed by typing the program name followed by `-h`. (e.g. `simhdr -h`).

Functional Extensions of Arroyo

Parallelization

Currently Arroyo itself contains no parallelization support. However, programmers are free to write parallelized programs that use Arroyo's functionality. One can attempt to parallelize by writing a multithreaded program to run on a multiprocessor machine or by writing a program to run on a cluster of workstations using a parallel programming library like MPI. One can also attempt to write a multithreaded parallel application to use on a cluster of multiprocessor workstations.

With a reasonable degree of restraint on the part of the programmer, Arroyo may be used in a

multithreaded application. Essentially one should be able to call the const member functions of a class instance from different threads. Certain difficulties exist because some Arroyo classes require memory space that is unallocated upon instantiation and is allocated later through certain member functions. I am currently looking into some ways to make this situation more thread safe, but I think experience will indicate what needs to be done.

Arroyo is capable of writing almost every class in the library to a fits file. This functionality permits a simple method of parallelizing the wave propagation across multiple nodes when one is not interested in simulating the AO control loop. One may have a bunch of processors read files containing an instance of a **refractive_atmospheric_model** and one or more instances of the **emitter** class. One could then write code to communicate to each processor the parameters of the simulation (e.g. how long it should last, the electromagnetic wavelengths to use in the simulation, etc.). Such code could be written using a number of parallel programming libraries, though I would choose to do this in MPI. Each processor can generate the same instance of the **refractive_atmospheric_layer** class by seeding the random number generator in the same way and by using identical arguments to the **refractive_atmospheric_model::get_refractive_atmospheric_layers** member function. In this way the resulting instances of the **refractive_atmospheric_layer** will be identical. The simulation could then be divided up among the processors in any number of ways, depending on the number of nodes in the cluster. An even cruder realization of this scheme could avoid the use of any parallel programming techniques by specifying the division of labor ahead of time in a file. One could then start a bunch of processes using a simple script that invokes rsh or something similar.

As Arroyo evolves to include support for AO systems, parallelization techniques will need to become more sophisticated. The existence of an AO control loop introduces a time dependence in the simulation. Wavefronts are corrected by one or more DMs before being detected by the WFS, and the WFS data is then used to change the DM surfaces. In this situation the strategy of parallelizing over time steps doesn't work. Instead, one may want to allocate some processors to perform the wave propagation from each emitter down to the wavefront sensors, while other processors reconstruct the wavefront phases and modify the DM surfaces. This division of labor requires synchronization between the processors, and load balancing will depend on the parameters of the simulation. It seems fairly clear that such simulations will be much easier if Arroyo adds functionality to transfer instances of classes between processors. Such functionality could be added on a class by class basis as needed.

GUI

For many applications a GUI is just a simple interface to a small number of underlying function calls. In these circumstances any GUI toolkit will do. For Arroyo the interface is much more demanding. A GUI element must typically instantiate a derived class and return a pointer to its base class. In this way the underlying code remains unpolluted by specific derived classes, and extensibility is maintained. The consequence of not choosing to mirror the inheritance hierarchy is to create an interface mismatch between the GUI and the library. As the project grows, this mismatch will cost more and more resources to maintain. This is the usual way in which projects die. Having made all the previously hidden data visible through this flat interface, programmers may be tempted to write software using the visible data rather than sticking with the API. This would be a disaster. It is much better to firmly wed the gui elements to their corresponding classes by mirroring the hierarchy.

To achieve this seamless use of derived classes, it is far more practical to use a GUI that actually supports C++ concepts like inheritance. GUI libraries like QT and gtkmm provide this level of functionality. Gtkmm has better C++ language support, but is at an earlier stage of development than is QT.

Because of the heavy reliance of this project on rather sophisticated C++ language features, I've decided on gtkmm rather than QT. With these libraries, one may write a gui class hierarchy that mirrors the underlying library. This class may inherit the library class through virtual inheritance. To date only a small amount of time has been spent on this - as stated above, I feel it would be a mistake at this point to sacrifice low level functionality for gui functionality at this stage of development. Some gtkmm gui code used to be included in the Arroyo distribution. However, these classes have now been moved to a different project that has yet to generate any stable code.

A GUI might allow astronomers to run simple test cases to see their objects using different designs. Simulations of a few seconds of simulation time on a 30 meter telescope are certainly feasible on a 2 GHz Pentium on a timescale of order hours. This would be a good PR tool.

Distribution and Support

Portability

Arroyo aims to cut down on development time by farming out as much functionality as possible to third party libraries. However, no proprietary 3rd party libraries or packages are being used. Furthermore, I have chosen 3rd party libraries that have themselves been compiled on a wide variety of platforms. To my knowledge Arroyo itself is platform independent, but has only been compiled on Red Hat linux OS. I expect Arroyo could easily be ported to Solaris, Dec OSF1, HP, SGI, other forms of the Linux OS like Debian, and 64 bit linux OS's for the Itanium processor. I am less certain about a Windows port, primarily because I've never programmed under Windows.

Revision control

Arroyo is under CVS, and is currently available to seven collaborators via remote checkout. There is a web based interface that one may use to browse the source code. There is also a mailing list for these collaborators. So far I am the only programmer contributing to library development, and when the functionality reaches a milestone I check in my code. If other people become interested in contributing to Arroyo's development, then I may have to rethink this approach. In the longer term I would like to have a stable branch for public release and a development branch to which new functionality is being added.

Licensing

The licensing situation for Arroyo is still evolving. I certainly plan to release Arroyo under the Gnu Public License. To date I have not made the effort to do so because the level of interest in the project has not justified such an effort. In passing I should note that I've made Arroyo available via CVS to all academic researchers that have expressed interest in getting the source code.

There is also an opportunity to release a commercializable version of Arroyo. This release would probably follow a model like that used by FFTW, where commercial users pay a relatively inexpensive one time as-is licensing fee for a version of Arroyo. I view this as an opportunity to expand the potential user base of Arroyo to include industrial as well as academic users, rather than as a potential source of income. The details of such an arrangement would need to be worked out with Caltech, and I probably wouldn't bother until there was a concrete expression of interest. I should note that there is no reason that industrial researchers need a commercial license to run simulations using Arroyo. However, selling

executables that link against a GPL version of Arroyo requires the seller to release the source code of this executable under the terms of the GPL. Please see the license for exact terms.

Distribution and Installation

The distribution uses Automake, Autoconf and Libtool. These GNU packages are used to construct a configure script based on the properties of your system. The script bootstrap.sh invokes these programs to create the script. Unfortunately, you actually need to install all of the third party libraries in order to run this script, because automake looks for m4 macros installed by these packages in constructing the configure script.

The configure script takes a number of options that determine the compilation strategy. The user may choose to support certain optional features of Arroyo, make certain decisions about third party libraries, or modify the installation directories. This script then generates the Makefiles. One may then type make and make install to compile and install the header files and shared object libraries. Arroyo also comes with a number of binaries that serve as simulation example programs, and which are installed as well.

When Arroyo is ready for public release, it will be available as a tarred, gzipped file. The configure script can be included in this file so that the step involving the bootstrap.sh script can be skipped. This will allow users to install only the third party libraries required to support the elements of Arroyo they want. For example, five libraries are required to support the optional GUI functionality. Installation of these libraries may be skipped if the user never intends to use the GUI.

Documentation

Overall documentation for the project is provided by this document. Arroyo uses the program Doxygen to document the API. If you have installed Doxygen, this documentation will automatically be generated in the make process. A version of this documentation is also available on my web page. This documentation consists of heavily cross-linked HTML pages with documentation imported directly from comments in the header files. This is a critically important bit of functionality - any design changes or additions to the header files are imported into the documentation when the user runs make.

Status and Future Plans

I have been working on this project for 1 year and 4 months. The first six months were mainly efforts to understand the problem of MCAO and to plan the API. Arroyo is approaching its conceptual halfway point. At this point it will be delivering statistically valid plane and spherical wavefronts to an aperture. One also has the option of performing a far field diffractive propagation to get the PSF. This is roughly half the development time, but only a small percentage of the research time and computer time that this project will use. Already at Caltech we have run simulations on a number of telescopes, emitter configurations, and atmospheres. Many of the results may be viewed on the Arroyo home page. To reach the halfway point, in the next couple of months I will be finishing up the spherical wave diffractive propagators and adding support to model the LGS uplink. I will also be adding verification tests. Upon completion of these items, I would like to branch in CVS and form a stable release, which I will publically distribute. I will then continue to develop the AO functionality on a development branch.

In the next stage of the project, I will aim to support a single conjugate AO system. This requires the following classes: detector, shack-hartmann wavefront sensor, deformable mirror, tip-tilt mirror, control

loop, and reconstructor. I may decide to postpone adding functionality to make the reconstructors, and instead use reconstructors provided by other researchers. This is partly because we would be able to start using Arroyo as a reconstructor testbed sooner and partly because other researchers may not be convinced of the results unless they provide their own reconstructor.

Future development depends on public response. The upside is that this is a solidly designed portable library project with a well-designed API, documentation, revision control, and distribution support. A downside is that this project has been written in C++, which is a language most people in the AO community do not use.

There is also a novel element to this software. Other simulations described above maintain their identity because they typically consist of a single executable. I went to a lot of effort to make this project a library rather than one or more executables because I felt that this would broaden the applicability of the project to a number of research areas. With a broad user base, this project is more likely to receive development support and contributions of software from its users. The downside to this strategy is that Arroyo risks losing its identity because each program that links against Arroyo becomes a GSMT simulation. Only time will tell whether support for this library is sustainable.

Acknowledgements

Development of Arroyo is funded by the California Institute of Technology for the development of adaptive optics systems on CELT.

Useful URLs

Simulations

Simulations of wave propagation through turbulence are usually written by military contractors or astronomers. The former typically consider their software and documentation to be proprietary, while the latter don't have time to post software or documentation on the web. Below are a list of links that point to software and/or documentation for some of the simulations mentioned above.

AOTools	www.tosc.com
Arroyo	eraserhead.caltech.edu
F. Rigaut's simulation	available from babcock.ucsd.edu/cfao_ucsd/software.html
Light Pipes	www.xs4all.nl/oko/pipes
Matlab Light Pipes	www.tn.utwente.nl/inlopt/lpmlab/index.htm
TASAT	www.northropgrummanit.com/tasat/index.html
ESO's simulation	www-obs.univ-lyon1.fr/tmr-lgs
Wave Train	www.mza.com/doc/wtug/index.htm
Zemax	www.zemax.com

AO related software

A++	available from babcock.ucsd.edu/cfao_ucsd/software.html
IDAC	babcock.ucsd.edu/cfao_ucsd/idac/idac_package/idac_index.html

Third Party Libraries and Packages used by Arroyo

Cfitsio	heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html
CVS	www.delorie.com/gnu/docs/cvs/cvs_toc.html
Doxygen	www.stack.nl/~dimitri/doxygen/index.html
FFTW	www.fftw.org
GCC	www.gnu.org/software/gcc/gcc.html
MPICH	www-unix.mcs.anl.gov/mpi/mpich

References

- Alexandrescu, 2001. “Modern C++ Design”, Addison-Wesley
- van Dam, M. A. & Lane, R. G. 2002, Applied Optics, 41, 5497.
- De Rose, T. 1989. University of Washington Department of Computer Science technical report UW-CSE-89-09-16.
- Dressler, A. *et al.*, 2001. In “Astronomy and Astrophysics in the New Millenium”, National Academy Press.
- Ellerbroek, B. L. 2002. JOSA A 19, 1803.
- Ellerbroek, B. L. & Cochran, G., 2002. Proc. SPIE 4494, 104.
- Ellerbroek, B. L. & Rigaut, F. J., 2000. Proc. SPIE 4007, 1088
- Goodman, J. W. 1993. “Introduction to Fourier Optics”, McGraw-Hill.
- Hardy, J. W. 1998. “Adaptive Optics for Astronomical Telescopes”, Oxford University Press.
- Le Louarn, M. 2002. MNRAS, 334, 865L.
- Lane, R. G., Glindemann, A. & Dainty, J. C., 1992. Waves in Random Media, 2, 209.
- Johansson, E. M. & Gavel, D. T. 1994. Proc. SPIE 2200, 372.
- Johnston, R. A. & Lane, R. G., 2000. Applied Optics, 39, 4761.
- Noll, R. J. 1976. JOSA, 66, 207.
- Papalexandris, M. V. & Redding, D. R., 2000. JOSA A, 17, 1763.
- Sasiela, R. J. 1994. “Electromagnetic Wave Propagation in Turbulence”, Springer-Verlag.
- Siggraph Conference Proceedings, 1989.
- Stoer, J. & Burlisch, R. 1993. “Introduction to Numerical Analysis”, Springer-Verlag.
- Stroustrup, B. 1997. “The C++ Programming Language”, 3rd edition. Addison-Wesley.
- Troy, M. *et al.* 2000. Proc. SPIE 4007, 31.
- Winker, D. M. 1991. JOSA A, 8, 1568.