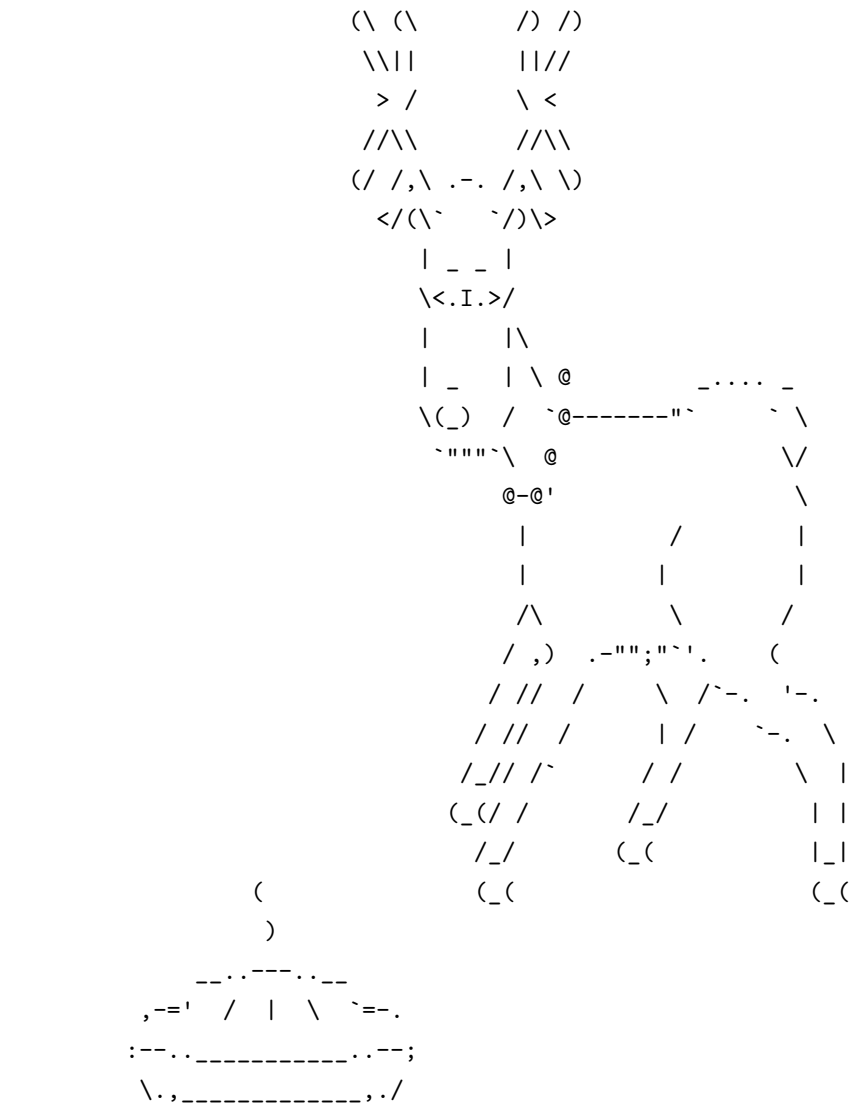
Core Loading Pattern Optimization

22.251 Final Project Report

Jeremy Roberts



("poro" is Finnish for reindeer)

Contents

CHAPTER 1

Introduction

1.1 Background

Nuclear fuel management is an important component of the broader nuclear fuel cycle, and it entails a number of choices that ultimately try to minimize costs while simultaneously meeting constraints imposed by the operator or regulator. Traditionally, fuel management has been somewhat arbitrarily divided into *out-of-core* and *in-core* fuel management [1]. The former involves many cycles and focuses on the fundamental problem of minimizing the levelized energy cost. Out-of-core fuel management decisions include deciding how much energy to produce, what fuel—new or replaced—should be used, among others. On the other hand, in-core fuel management focuses on how the available fuel should be used, including its position and alignment, in addition to the type and placement of burnable poisons, and, for a BWR, the control rod programming. Moreover, each of these variables must be considered along with safety constraints and the overarching goal of economics.

Obviously, the whole of nuclear fuel management is an enormous opti-

mization problem—so large that comparatively little work has been done historically to couple the out-of-core and in-core components even though they are decidedly a very coupled system. Typically, the out-of-core aspects have been treated with rather coarse methods such as the linear reactivity model [2]. Such methods are extremely fast, and for gross characteristics of material flows and economics, they offer reasonable insight. For in-core aspects, a more detailed physics analysis is required, usually accomplished via coarse mesh two-group or one-and-a-half group diffusion solvers. These are quite fast and reasonably accurate methods but have historically been too expensive to be used within the broader out-of-core analysis given the shear magnitude of the solution space.

Recently, more work has been done to optimize over multiple cycles, using increasingly powerful computers and highly parallelized optimization tools [3]. Such work suggests that automated optimization techniques may find increasing production use by utilities and other fuel management stakeholders.

1.2 A Simple Tool

The general goal of this and other projects is to provide a *fast* and *flexible* tool for analyzing a particular component of the fuel cycle. To be *fast*, the tool (probably) should sacrifice some accuracy for speed. To be *flexible*, the tool should be **free** and **open source** as well as created in an **easy-to-use** setting with **thorough documentation**. While there are a handful of tools for single- or multi-cycle loading pattern optimization available, none has all the features desired for this project. In particular, none of the tools appears to be free (or open source) or they were introduced so long ago as academic tools that they likely do not exist today. Table 1.1 lists several of these tools, a few of which are well-established. These are just a sample of tools encountered in the literature; many more, especially academic, likely

exist. Table 1.1 provides the code name (with an appropriate reference), the authoring institution, and some information regarding the solver, optimization method (genetic algorithm (GA) or simulated annealing (SA)), and if it is available commercially or in some other form.

Code	Author	Notes
XIMAGE[4]	Studsvik	SIMULATE as solver; SA; commercial
COPERNICUS[3]	Studsvik	SIMULATE as solver; parallel SA; commercial
ROSA[5]	NRG	1.5 group LWRSIM as solver; SA; commercial
FORMOSA-P/B[6, 7]	NCSU	Generalized perturbation theory; SA; academic, but not free
CIGARO[8]	Penn State	SIMULATE as solver, though general; GA; academic, status unknown
SOPRAG[9]	IIE (Mexico)	PRESTO-B as solver; SA and GA; research, status unknown
FuelGen[10]	U. of Greenwich	Solver unknown; GA; academic, status unknown
XCore[11]	TAEK (Turkey)	Neural net as solver; GA; research, status unknown
GARCO–PSU[12]	Penn State	SIMULATE as solver; GA; academic, status unknown

Table 1.1: Representative collection of core loading pattern design tools.

Alternatively, it may be possible to combine various open source tools into a single optimization framework. Probably the most useful component would be a free and open source core analysis code; unfortunately, there appear to be relatively few applicable codes and fewer free ones. The fun-

damental limitation to any such code would be flexibility: the class tools are almost certainly best served if they rely on home grown components. That way, the components will be simple and well-understood.

Consequently, for this project a simple core loading pattern tool is proposed. The tool will initially be limited to two-dimensional patterns, essentially limiting analyses to PWR's. Additionally, only beginning-of-cycle studies will be considered, as incorporation of burnup and other factors will depend on tools provided by other projects.

1.3 Overview of `poropy`

Core Physics

For the tool to be successful, it needs a fast neutronics solver and an effective optimization scheme. As a crude benchmark, we note that in 2003, the ROSA tool performed full cycle evaluations for a single pattern in as few as 25 ms for 10-20 burnup steps [5]. Hence, a single time step was computed in a few ms. In the 8 years since then, Moore's law suggests we should see 4- or 5-fold reduction in that time, or about 0.1 ms, a really small number. Of course, Moore's law stopped applying to individual CPU's some time ago, and so it is necessary to include multicore computing in that estimate. For a new 6-core machine with hyperthreading, one can imagine solving a single state in a few ms, and over all (hyper)threads, the total wall time is reduced to the the tenths of ms range. Hence, for single CPU's, the nominal goal for this project is about 1 ms. Neglecting all other considerations, this timing yields an overall goal of hundreds of millions of time steps per day. The basic goal used as a guide (and motivator) throughout is to be able to run 200 million patterns per day using any parallelism available on a powerful desktop machine.

For this project, the author's own response matrix code `serment` was originally selected. Currently, `serment` uses fine mesh finite difference dif-

fusion to compute "response functions", which are nothing more than outgoing partial currents from each surface or volume-integrated reaction rates due to an incident partial current on a particular surface. This is a relatively expensive method when the number of unique assemblies is large, as is the case for realistic studies including burnup. As a cheaper alternative, a simple semi-analytic model for two-group responses was developed and improved by nonlinear regression. The chief goal was to reproduce fine mesh responses for the zeroth other (*i.e.* flat partial current) approximation within one percent. The zeroth order approximation is rather coarse, achieving RMS (MAX) relative errors in fission rates of about 10 and 30 percent for common benchmark problems. However, it is a rather efficient approximation, and by using a simple response function model, it is quite suitable for the optimization scheme. Chapter 2 provides an overview of the response matrix method and gives a detailed account of the semi-analytic response function model.

Unfortunately, two issues made use of `serment` less favorable. First, `serment` relies on several third party libraries that introduced nontrivial changes in their most recent releases that have led to a few performance issues that need to be resolved. Second, `serment` is written in C++, and to generate the requisite Python bindings would have taken valuable time from the focus of this work. As an alternative, the LABAN-PEL code is used [13]. LABAN-PEL is a response matrix code for multigroup diffusion based on the dissertation of Lindahl [14] and extended by Müller at Pelindaba in South Africa [15]. The code is relatively fast (in the ms range per time step) and has an easy input and output. No time was spent to interface with LABAN-PEL directly, using simple Python scripting to read and write the text input and output files. The response matrix discussion of Chapter 2 also applies to LABAN-PEL, and its input format is presented by example.

Optimization

For the optimization, the approach of genetic algorithms (GA) was selected. In particular, the library PGAPack [16] was selected, due to the author's limited past experience. PGAPack is a parallel implementation of GA written in C. However, for this project to be flexible, it was decided early on to use Python for the final product. As a result, it is necessary to access the various libraries directly used via Python, and so a Python interface for PGAPack called `pypgapack` was written with the help of SWIG and can be found online [17] with full documentation [18]. Importantly, `pypgapack` maintains all the parallel features of PGAPack, thus increasing significantly the number of evaluations possible for optimization. Chapter 3 gives a very brief summary of genetic algorithms and suggests several references for more information. Additionally, several examples can also be found in the online documentation of `pypgapack` that help illustrate various points.

For core loading patterns, specialized genetic operators are required within the GA to ensure possible solutions are valid. For this project, Poon's Heuristic Tie-Breaking Crossover (HTBX) operator [19] was implemented. The essential idea of the heuristic is that children are in some sense neutronically similar to their parents. This and the related (but simpler) Tie-Breaking Crossover (TBX) operator are discussed in Chapter 3. The TBX operator is applied to a simple Traveling Salesman Problem. The HTBX operator is applied to a simple one-dimensional, two-group slab reactor problem, where the goal is to maximize k and minimize power peaking, and where a reference solution was found by exhaustive search.

Finally, the `pypgapack` and a set of reactor-specific Python tools were combined in a new package called Physics Of Reactors Optimization in PYthon (`poropy`). The code is organized in object-oriented format. Simulations are represented in terms of a `Reactor` object, which has an associated inventory of `Assembly` objects that reside in the core or in the spent fuel pool. In an interactive Python session, the user can manually shuffle

bundles using several commands, following which the power map can be displayed and other information can be printed. For an automated optimization process, the user can use GA with HTBX. Chapter 4 gives a more detailed account of the code and the results of several test cases. The code and documentation are not yet online, but will be as time allows.

CHAPTER 2

Neutronics

For core loading pattern optimization, one needs an efficient, relatively accurate physics model. Most core analysis uses the few group diffusion approximation. In general, solution by a fine mesh finite difference approach is too costly. However, a number of much faster methods exist. These include several flavors of nodal methods, including the "modern" or "consistently formulated" analytic and polynomial nodal methods, as reviewed in Ref. [20], and more historical nodal methods such as FLARE and PRESTO, as reviewed in Ref. [21]. Also possible are coarse mesh finite difference or low order finite element approximations, which, while fast, are generally far less accurate than the nodal schemes for a given computational cost.

An alternative approach used here is the eigenvalue response matrix method, where coarse meshes are linked by (partial) currents, and where the current response to an incident current is an implicit function of the eigenvalue k. To evaluate these responses, various transport approximations have been used, ranging from finite difference diffusion to Monte Carlo transport. We first propose a simple semi-analytic diffusion model

for zeroth order response functions, similar in some sense to low order analytic nodal methods. The model can be used as is or with parameters from a fit to reference fine mesh data for improved accuracy.

Due to time and other constraints, the response matrix code LABAN-PEL [13] is used for the analysis in Chapter 4, and a brief overview of relevant input parameters is provided below by example.

2.1 Response Matrix Method

The response matrix method (RMM) has been used in various forms since the early 1960's [22]. Using the terminology of Lindahl and Weiss [23], the method can be formulated using explicit volume flux responses, called the "source" RMM, or by using current responses that include fission implicitly and hence are functions of k, known as the "direct" RMM. While both methods are used in various nodal methods, the former is more widespread; this work employs the latter, which shall be referred to as the *eigenvalue* response matrix method (ERMM).

For time independent core analyses, one solves the transport equation, represented compactly as

$$\mathbb{T}\phi(\vec{\rho}) = \frac{1}{k}\mathbb{F}\phi(\vec{\rho})\,, \tag{2.1}$$

where the operator $\mathbb{T}$ describes transport processes, $\mathbb{F}$ describes neutron generation, ϕ is the neutron flux, $\vec{\rho}$ represents the relevant phase space, and k is the eigenvalue, the ratio of the number of neutrons in successive generations.

Suppose the global problem of Eq. 2.1 is defined over a volume V. Then a local homogeneous problem can be defined over a subvolume V_i subject to

$$\mathbb{T}\phi(\vec{\rho}_i) = \frac{1}{k}\mathbb{F}\phi(\vec{\rho}_i)\,, \tag{2.2}$$

and

$$J_-^{\text{local}}(\vec{\rho}_{is}) = J_-^{\text{global}}(\vec{\rho}_{is})\,, \tag{2.3}$$

where $J_{-}^{\text{local}}(\vec{\rho}_{is})$ is the incident neutron current (a function of the boundary flux) on surface s of subvolume i.

To treat the problem numerically, boundary currents are expanded in an orthogonal basis set of dimension N, which in multidimensional models is comprised of tensor products of orthogonal sets corresponding to each phase space variable. For the case of multigroup diffusion, a natural approach is to keep the multigroup approximation for energy, and use a truncated set of Legendre polynomials for the spatial dependence. More formally, the outgoing/incident current in group g from horizontal side s of subvolume i is expanded as

$$J_{igs\pm}(x) \approx \sum_{n=0}^{N} j_{i\pm}^{gns} P_n(2x/\Delta)\,,$$

and

$$j_{i\pm}^{gns} = \frac{2n+1}{2} \int_{-\Delta/2}^{\Delta/2} P_n(2x/\Delta) J_{is\pm}(x) dx\,, \tag{2.4}$$

where P_n is the nth order Legendre polynomial.

Working completely with these expanded variables, the group-wise Legendre moments on each of S subvolume surfaces are guessed, and the outgoing current is computed in terms of *response functions*, thus setting up an iterative procedure. While response functions are given a formal definition below, it is worth noting they are similar in concept to reflection and transmission coefficients, quantities often computed by hand in textbook analyses of monoenergetic slab problems.

In response matrix form, the current balance in a square mesh i for two

groups and an Nth order Legendre expansion is

$$\mathbf{J}_{i+} = \begin{bmatrix} j_{i+}^{101} \\ j_{i+}^{111} \\ \vdots \\ j_{i+}^{1N1} \\ j_{i+}^{201} \\ \vdots \\ j_{i+}^{2N4} \end{bmatrix} = \begin{bmatrix} r_{i101}^{101} & r_{i101}^{111} & \cdots & r_{i101}^{1N1} & r_{i101}^{201} & \cdots & r_{i101}^{2N4} \\ r_{i111}^{101} & r_{i111}^{111} & \cdots & r_{i111}^{1N1} & r_{i111}^{201} & \cdots & r_{i111}^{2N4} \\ & & \ddots & & & & \\ r_{i1N1}^{101} & r_{i1N1}^{111} & \cdots & r_{i1N1}^{1N1} & r_{i1N1}^{201} & \cdots & r_{i1N1}^{2N4} \\ r_{i201}^{101} & r_{i201}^{111} & \cdots & r_{i201}^{1N1} & r_{i201}^{201} & \cdots & r_{i201}^{2N4} \\ & & \ddots & & & & \\ r_{i2N4}^{101} & r_{i2N4}^{111} & \cdots & r_{i2N4}^{1N1} & r_{i2N4}^{201} & \cdots & r_{i2N4}^{2N4} \end{bmatrix} \begin{bmatrix} j_{i-}^{101} \\ j_{i-}^{111} \\ \vdots \\ j_{i-}^{1N1} \\ j_{i-}^{201} \\ \vdots \\ j_{i-}^{2N4} \end{bmatrix}$$
$$= \mathbf{R}_i \mathbf{J}_{i-} \,, \tag{2.5}$$

where $\mathbf{J}_{i\pm}$ is the vector of outgoing/incident partial current expansion coefficients defined

$$j_{i\pm}^{gns} \equiv \quad n\text{th Legendre moment of the } g\text{th group outgoing/incident partial current on/from side } s \text{ of subvolume } i \quad ,$$

and $\mathbf{R}_i$ is the matrix of response functions defined

$$r_{igns}^{g'n's'} \equiv \quad n\text{th Legendre moment of the outgoing partial current in the } g\text{th group and from side } s \text{ of subvolume } i\text{, due to an incident partial current of unit strength in group } g'\text{, incident on side } s'\text{, and whose shape is the } n'\text{th Legendre polynomial} \quad .$$

The corresponding global equations can be expressed as

$$\begin{aligned} \mathbf{J}_+ &= \mathbf{R}(k)\mathbf{J}_- \\ \mathbf{J}_- &= \mathbf{M}\mathbf{R}(k)\mathbf{J}_- \,, \end{aligned} \tag{2.6}$$

where $\mathbf{R}(k)$ is a block diagonal matrix of the $\mathbf{R}_i$ and $\mathbf{M} = \mathbf{M}^T$ redirects the outgoing $\mathbf{J}_+$ of one subvolume as an incident $\mathbf{J}_-$ of an adjacent subvolume.

To solve this global problem, one iterates on

$$\mathbf{J}_{-}{}^{(\text{new})} \leftarrow \mathbf{MR}(k)\mathbf{J}_{-}{}^{(\text{old})} , \tag{2.7}$$

with updates to k defined as

$$k^{(\text{new})} = \frac{\mathbf{F}(k^{(\text{old})})\mathbf{J}_{-}}{(\mathbf{A}(k^{(\text{old})}) + \mathbf{L}(k^{(\text{old})}))\mathbf{J}_{-}} , \tag{2.8}$$

where $\mathbf{F}$ yields the global fission rate, $\mathbf{A}$ yields the global absorption rate, and $\mathbf{L}$ yields the net leakage rate at global boundaries.

2.2 Baseline Model

As an alternative to generating responses by a fine mesh finite difference approach, a simple semi-analytic model is developed for the two group diffusion equations in a homogeneous square region. For two groups and an Nth order spatial expansion, and assuming four-way symmetry (so the incident side index is omitted), Table 2.1 lists the number of individual response functions needed for a single coarse mesh. The addition responses $F^{g'0}$ and $A^{g'0}$ are the volume-integrated fission and absorption rates due to a flat current in group g' and make up the F and A operators of Eq. 2.8. The response $\Phi_g^{g'}$ is the volume-averaged group g flux due to a flat current in group g'. Note that higher order terms do not introduce net reaction rates (or fluxes) in homogeneous coarse meshes due to symmetry. While $\Phi_g^{g'}$ is not strictly needed, it is a useful quantity to have. For this project, only zeroth order responses are considered, which as mentioned above corresponds to the flat or average current approximation. Scoping studies suggest the work described below will apply equally well to first order responses, but the volume of work getting there was outside the intended scope.

To determine the responses, the two group diffusion equation must be solved. Assuming fission neutrons are born only in the fast group and ne-

order:	0	1	2
current responses, $r_{gns}^{g'n'}$	12	48	108
fission responses, $F^{g'}$	2	2	2
absorption responses, $A^{g'}$	2	2	2
total group responses, $\Phi_g^{g'}$	4	2	2
TOTAL	20	54	114

Table 2.1: Numbers of needed response data by spatial order.

glecting upscatter, the two group equations are

$$\begin{aligned} -D_1\Big(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\Big)\phi_1(x,y)+\Sigma_{R1}\phi_1(x,y) &= \frac{1}{k}\Big(\nu\Sigma_{F1}\phi_1(x,y)+\nu\Sigma_{F1}\phi_2(x,y)\Big) \\ -D_2\Big(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\Big)\phi_2(x,y)+\Sigma_{A1}\phi_2(x,y) &= \Sigma_{12}\phi_1(x,y)\,, \end{aligned} \tag{2.9}$$

subject to appropriate boundary conditions.

These equations are coupled in energy. We can decouple them by using the so-called *linear transformation technique*, following Hebert [24]. We first rewrite Eq. 2.9 as

$$\Big(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\Big)\phi_g(x,y)+H_g\phi_g(x,y)=0\,, \tag{2.10}$$

where H_g is the gth row of the matrix $\mathbf{H}$, the elements of which are defined

$$H_{gg'}=\frac{1}{D_g}\Big(-\Sigma_{Rg}\delta_{gg'}+\Sigma_{g'g}(1-\delta_{gg'})+\frac{\chi_g}{k}\nu\Sigma_{Fg'}\Big)\,. \tag{2.11}$$

Define $\mathbf{V}$ to be the matrix whose columns are the eigenvectors v_g of $\mathbf{H}$ with associated eigenvalues λ_g. Then

$$\mathbf{HV}=\mathbf{V\Lambda}\,, \tag{2.12}$$

where $\mathbf{\Lambda}$ is a diagonal matrix with elements λ_g. We then define *modal* fluxes ψ_g implicitly as

$$\phi_g(x,y) = V_g \psi_g(x,y)\,, \tag{2.13}$$

where V_g is the gth row of $\mathbf{V}$. Substituting this form for ϕ_g into Eq. 2.10 yields

$$\Big(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\Big)\psi_g(x,y) + \lambda_g \psi_g(x,y) = 0\,. \tag{2.14}$$

Note, Eqs. 2.10-2.14 are completely general and apply to any number of groups. In one dimension, the transformation to "modal" space allows analytic solutions for any problem up to four energy groups. Beyond four groups, there is no general analytic solution for the eigenpairs of $\mathbf{H}$, but the eigenproblem can be solved numerically with very high accuracy, rendering the solution essentially exact.*

The modal fluxes $\psi_g(x,y)$ are still two-dimensional. As is common in semi-analytic methods, we reduce the problem to one dimension. The approach taken here is influenced by Lindahl's semi-analytic approach, which makes use of Fourier expansions for both x and y but treats the the transverse direction differently [14]. Here, we limit the transverse expansion to just one even term, a cosine. In other words, for an incident current on the left side of a square node, the transverse (y-directed) modal flux is assumed to be of the form

$$Y_g(y) = n_g \cos(\nu_g y)\,, \tag{2.15}$$

subject to

$$n_g^{-1} = \int_{-\Delta/2}^{\Delta/2} Y_g(y)dy\,, \tag{2.16}$$

and

$$\frac{1}{4}Y_g(\pm\Delta/2) \pm \frac{D_g}{2}\frac{\partial}{\partial y}Y_g(y)\Big|_{y=\pm\Delta/2} = 0\,, \tag{2.17}$$

*Recall that an eigenproblem of dimension n is equivalent to finding the roots of an associated characteristic polynomial of degree n. By the Abel-Ruffini theorem[25], we know the roots of polynomials of degree 5 or higher cannot generally be found analytically; rather, iterative schemes are needed to generate approximations.

the zero incident current condition. Note, this is really an arbitrary condition, since $Y(y)$ is in *modal* space, whereas the current condition implies a *physical* flux. Even so, the form has worked fairly well. The parameter ν_g is found by solving the transcendental equation

$$\frac{1}{2D_g}\cot\left(0.5\nu_g\right) = -\nu_g\,, \tag{2.18}$$

which by a truncated Taylor series yields

$$\nu_g \approx \left(\frac{0.25/\Delta - \varepsilon}{0.020833\Delta + 0.25D_g}\right)^{1/2}, \tag{2.19}$$

where ε replaces the right hand side of Eq. 2.17. It turns out the (unfitted) model works slightly better if ε is a small negative number, which has the effect of increasing the leakage. Reasonable values were -0.001 and -0.005 for groups 1 and 2, respectively; this effect is automatically accounted for in the fitted model discussed below.

Assuming that $\psi_g(x,y) = \psi_g(x)Y(y)$ and inserting into Eq. 2.14 yields

$$\begin{aligned}
\frac{\partial^2}{\partial x^2}\psi_g(x)Y(y) + \frac{\partial^2}{\partial y^2}\psi_g(x)Y(y) + \lambda_g\psi_g(x)Y(y) &= 0\\
\frac{\partial^2}{\partial x^2}\psi_g(x)Y(y) - \nu_g^2\psi_g(x)Y(y) + \lambda_g\psi_g(x)Y(y) &= 0\\
\frac{\partial^2}{\partial x^2}\psi_g(x) + (\lambda_g - \nu_g^2)\psi_g(x) &= 0\\
\frac{\partial^2}{\partial x^2}\psi_g(x) - \kappa_g^2\psi_g(x) &= 0\,,
\end{aligned} \tag{2.20}$$

where $\kappa_g = \lambda_g - \nu_g^2$. Solutions of Eq. 2.20 take the general form

$$\psi_g(x) = a\cos(\sqrt{\kappa_g}x) + b\sin(\sqrt{\kappa_g}x)\,, \tag{2.21}$$

for positive κ_g and

$$\psi_g(x) = a\cosh(\sqrt{-\kappa_g}x) + b\sinh(\sqrt{-\kappa_g}x)\,, \tag{2.22}$$

for negative κ_g.

For completeness, we give explicit definitions for each of the two group equations and quantities of interest. The matrix $\mathbf{H}$ is defined

$$\mathbf{H} = \begin{bmatrix} \frac{\nu\Sigma_{F1}/k - \Sigma_{R1}}{D_1} & \frac{\nu\Sigma_{F2}}{D_1 k} \\ \frac{\Sigma_{12}}{D_2} & -\frac{\Sigma_{A2}}{D_2} \end{bmatrix},$$

and has eigenvalues

$$\lambda_{1/2} = \frac{\nu\Sigma_{F1}/k - \Sigma_{R1}}{2D_1} - \frac{\Sigma_{A2}}{2D_2} \pm \sqrt{\left(\frac{\nu\Sigma_{F1}/k - \Sigma_{R1}}{2D_1} - \frac{\Sigma_{A2}}{2D_2}\right)^2 + \frac{\nu\Sigma_{F2}\Sigma_{12}}{D_1 D_2 k^2}}, \tag{2.23}$$

and eigenvectors

$$\mathbf{V} = \begin{bmatrix} \lambda_1 + \frac{\Sigma_{A2}}{D_2} & \lambda_2 - \frac{\Sigma_{A2}}{D_2} \\ \frac{\Sigma_{12}}{D_2} & \frac{\Sigma_{12}}{D_2} \end{bmatrix}.$$

Note, the eigenvectors are unique only in direction, meaning they can be scaled arbitrarily; the definition given is a particularly compact one.

Then the group fluxes are defined (for the frequently observed case of $\kappa_1 > 0$ and $\kappa_2 < 0$)

$$\begin{aligned} \phi_1(x) &= V_{11}\Big(a\cos(\sqrt{\kappa_1}x) + b\sin(\sqrt{\kappa_1}x)\Big) + \\ &\qquad V_{12}\Big(c\cosh(\sqrt{-\kappa_2}x) + d\sinh(\sqrt{-\kappa_2}x)\Big) \\ \phi_2(x) &= V_{21}\Big(a\cos(\sqrt{\kappa_1}x) + b\sin(\sqrt{\kappa_1}x)\Big) + \\ &\qquad V_{22}\Big(c\cosh(\sqrt{-\kappa_2}x) + d\sinh(\sqrt{-\kappa_2}x)\Big), \end{aligned} \tag{2.24}$$

subject to the incident current conditions

$$\begin{aligned}
\frac{1}{4}\phi_1(0) - \frac{D_1}{2}\frac{\partial}{\partial x}\phi_1(x)\Big|_{x=0} &= \delta_{g'1} \\
\frac{1}{4}\phi_1(\Delta) + \frac{D_1}{2}\frac{\partial}{\partial x}\phi_1(x)\Big|_{x=\Delta} &= 0 \\
\frac{1}{4}\phi_2(0) - \frac{D_2}{2}\frac{\partial}{\partial x}\phi_2(x)\Big|_{x=0} &= \delta_{g'2} \\
\frac{1}{4}\phi_2(\Delta) + \frac{D_2}{2}\frac{\partial}{\partial x}\phi_2(x)\Big|_{x=\Delta} &= 0\,,
\end{aligned} \tag{2.25}$$

where g' is the group of the incident source. Equation 2.25 provides four conditions for four unknown coefficients a, b, c, and d. For each of two possible incident currents, this system is solved, giving the fluxes from which the required response function of Table 2.1 can be computed.

Computing the transverse leakage can be done in several ways. The one used here is simply to enforce balance. Defining the average outgoing partial current in group g from side s due to an incident current in group g' as $r_{gs}^{g'}$ for $s = l$, $s = r$, or $s = t$, for left, right, and transverse, and a corresponding volume-integrated flux as $\Phi_g^{g'}$, we have

$$\begin{aligned}
\delta_{g'1} - r_{1l}^{g'} - r_{1r}^{g'} - 2r_{1t}^{g'} + \Phi_1^{g'}(\Sigma_{F1}/k - \Sigma_{R1}) + \Phi_2^{g'}\Sigma_{F2}/k = 0 \\
\delta_{g'2} - r_{2l}^{g'} - r_{2r}^{g'} - 2r_{2t}^{g'} + \Phi_1^{g'}\Sigma_{12} - \Phi_2^{g'}\Sigma_{A2} = 0\,.
\end{aligned} \tag{2.26}$$

The transverse leakage in each group is then

$$\begin{aligned}
r_{1t}^{g'} &= 0.5(\delta_{g'1} - r_{1l}^{g'} - r_{1r}^{g'} + \Phi_1^{g'}(\Sigma_{F1}/k - \Sigma_{R1}) + \Phi_2^{g'}\Sigma_{F2}/k) \\
r_{2t}^{g'} &= 0.5(\delta_{g'2} - r_{2l}^{g'} - r_{2r}^{g'} + \Phi_1^{g'}\Sigma_{12} - \Phi_2^{g'}\Sigma_{A2})\,.
\end{aligned} \tag{2.27}$$

2.3 Fitted Model

While the model just described works surprisingly well, the goal is to match reference zeroth order core calculations to within one percent. Scoping studies suggested that individual response functions need to be within about 0.001 of the reference value to meet the goal. In other words, for a single

incident neutron, each response must add or subtract no more than 0.001 neutron, a relatively small fraction. (Of course, balance is *always* preserved by the definition of the transverse term).

Fuel Assembly Model

For fuel assemblies, the model used consists of the new forms for the buckling terms ν_g, defined

$$v_1 = \sqrt{\frac{B_1/\Delta + 0.001B_2}{B_5\Delta + B_6D_1}} \tag{2.28}$$

and

$$v_2 = \sqrt{\frac{B_3/\Delta + 0.020B_4}{B_7\Delta + B_8D_1}}\,. \tag{2.29}$$

Worth noting is that D_1 worked better in both ν_g terms, which is interesting because Lindahl suggested using the largest D value for a similar approach that used just a single buckling form for both groups [14]. While untested, it is likely the case that v_1 and v_2 could be replaced by a single value at little cost in accuracy.

Beyond the buckling form, which seemed to handle most gross dependencies, several individual response fixes are given in Table 2.2.

$r_1^{1l} = r_1^{1l} + B_9\Delta + B_{10}\lambda_1$	$r_2^{1l} = r_2^{1l} - 0.00482 + B_{11}\lambda_1$
$r_1^{1r} = r_1^{1r} + 0.006 + B_{12}\lambda_1$	$r_2^{1r} = r_2^{1r} + 0.00025 + B_{13}\lambda_1$
$r_1^{2l} = r_1^{2l} - 0.043 + B_{14}\lambda_1$	$r_2^{2l} = r_2^{2l} + B_{15}\Delta + B_{16}\lambda_2$
$r_1^{2r} = r_1^{2r} + 0.003$	
$A^1 = A^1 + B_{17}\lambda_1$	$A^2 = A^2 + B_{18}\lambda_1$
$F^1 = F^1 - 0.082 + B_{19}\lambda_1$	$F^1 = F^2 + 0.038 + B_{20}\lambda_1$

Table 2.2: Individual response function fitted corrections.

In many of the forms, somewhat arbitrary constant values appear. Most of these were found by manual iteration as additive fit parameters and then kept constant for further tweaking. This helped to reduce the search space.

To fit the model parameters B_i, reference response function data was generated by fine mesh diffusion, using a 100 by 100 grid. Two assembly widths of 20 and 22 cm were used, which bound most applicable assembly sizes. Eigenvalue values of 0.98, 0.99, 1.00, 1.01, and 1.02 were used. The two group data used as input came from several CASMO runs taken from previous class assignments. In particular, 559 state points spanning several burnups and assembly types were used, and the minimum and maximum of each constant is given in Table 2.3.

	minimum	maximum
D_1	1.4132e+00	1.5289e+00
D_2	3.4323e-01	4.0309e-01
Σ_{r1}	2.3885e-02	2.9202e-02
Σ_{a2}	9.0956e-02	1.2824e-01
$\nu\Sigma_{f1}$	3.8048e-03	7.9790e-03
$\nu\Sigma_{f2}$	9.2575e-02	1.8065e-01
Σ_{12}	1.3186e-02	1.7194e-02

Table 2.3: Minimum and maximum two group constant values.

To fit the model, MATLAB's nonlinear regression tool `nlinfit` was used. Initially, individual responses were fitted, but worked best in the end was to use a weighted sum of the absolute values of the residuals of all responses. In this way, the weights could be tuned to lower the maximum error found for a given parameter. If too little freedom existed (so that satisfying the 0.001 constraint on one parameter led another to violate it), another parameter was added. Table 2.4 gives the final values for the fit parameters.

parameter	value
B_1	0.261513147949597
B_2	0.014430493936852
B_3	-1.963269877409992
B_4	-1.686058019167120
B_5	0.020446975313514
B_6	0.211914109163937
B_7	0.006730851120201
B_8	-1.413949422121088
B_9	-0.000148580985696
B_{10}	0.612133597507877
B_{11}	0.069427926629857
B_{12}	0.562848227400144
B_{13}	0.029221824260804
B_{14}	-1.425806818665860
B_{15}	0.000910188034673
B_{16}	0.059643780247241
B_{17}	21.962969406904719
B_{18}	-2.368849865957258
B_{19}	21.640984894797850
B_{20}	-10.881107307101317

Table 2.4: Fitted model parameters.

Reflector Model

For the reflector, several benchmark reflector materials were used to generate response as a function of assembly width (from 4 to 24 cm) and the eigenvalue (same range as above). Included were the reflectors from the IAEA and Biblis benchmarks. MATLAB's curve fitting tool was used, and in all cases, each response was fit to a "22" rational fit, which is a five parameter fit of the form

$$f(x) \approx \frac{p_1 x^2 + p_2 x + p_3}{x^2 + + p_4 x + p_5}\,. \tag{2.30}$$

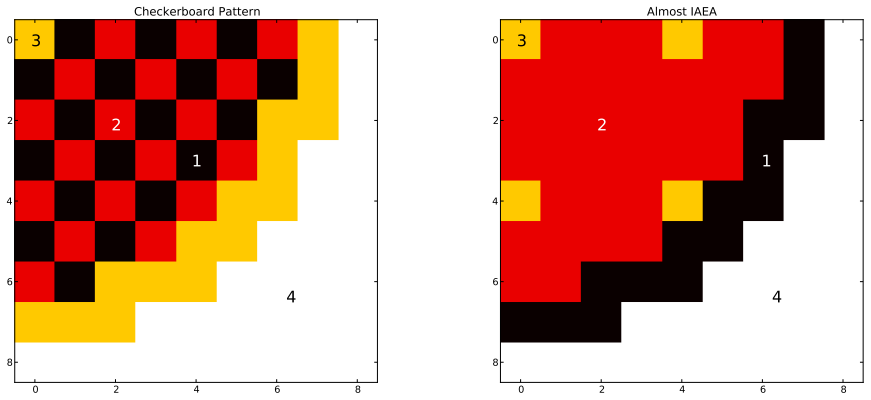
In all cases, the fits were extremely accurate ($R^2 \approx 1$). Because the tabulated data is relatively numerous, it is left out.

2.4 Numerical Study

With respect to the reference data, the model and fitted model yield maximum RMS (MAX) absolute errors of about 0.04 (0.06) and 0.002 (0.007) across all response functions. The 0.002 RMS error for the fitted model is a bit higher than desired, but as shown below, the fitted model does meet the goal of matching the fine mesh core results within one percent. It must be reiterated that the reference is a zeroth-order response matrix solution using fine mesh finite difference to generate responses.

To test the fits, two simple cases are used, illustrated in Figure 2.1. The first case uses data from the CASMO reference cases representative of fresh, once burned, and twice burned fuel, and uses the Biblis reflector material. The assemblies are 21 cm in width. For the second case, the IAEA group constants are used. Each assembly is 20 cm in width. In all cases, a reflective condition is used (due to a code limitation), which is not physical since it doubles the width of assemblies along the reflective boundary. For studying the numerical properties of the model, however, the cases should suffice.

For the checkerboard problem, the model achieved surprisingly good results for the fission density, less than 3 percent maximum. The error in the eigenvalue was $\epsilon_k = 169$ pcm. The fitted model was substantially better, being within about 0.2% on fission density and having $\epsilon_k = 10$ pcm. With respect to time, the reference solutions took on the order of tens of seconds, but note the underlying finite difference solver is not really optimized. For comparison, the model took just tenths of second (with the fitted model being slightly slower). For reference, the relative fission density error is given in Figure 2.2.



(a) Checkerboard (b) "Almost" IAEA

Figure 2.1: Simple test cases.

For the "almost" IAEA problem, the model achieved was within about 4 percent on fission density, with $\epsilon_k = -441$ pcm. The fitted model was within about 2% on fission density and $\epsilon_k = -211$ pcm. The timing trends match those for the checkerboard problem. The relative fission density error is given in Figure 2.3. What should be noted is that the IAEA fuel data does *not* fall within the range of CASMO reference values (but the reflector is still treated essentially exactly). Hence, while the fitted model does do better than the unfitted model, it does worse for this case than for the first problem.

2.5 Comment

While the fitted model does exactly what asked of it, perhaps it is worth revisiting the initial goal: is zeroth order actually good enough? While it is likely fast enough, it was mentioned (and can be confirmed in Ref. [13]) that zeroth order calculations yield relative errors in the fission density up

to 40 percent. Going to first order brings this to a more reasonable level, around 7 percent in the worst case in Ref. [13] but closer to 1 or 2 percent on average. Hence, it is probably worthwhile to pursue a semi-analytic approach for first order responses, and initial scoping studies suggest it's possible by adding only a single odd term akin to Eq. 2.15.

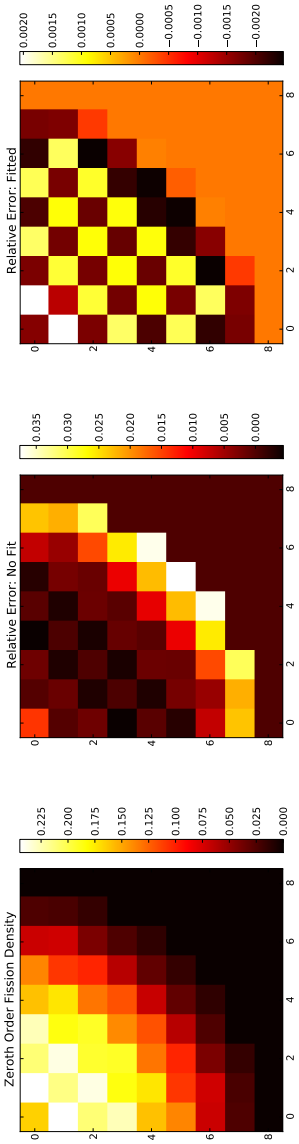


Figure 2.2: Reference (100x100 mesh) fission density with model relative errors.

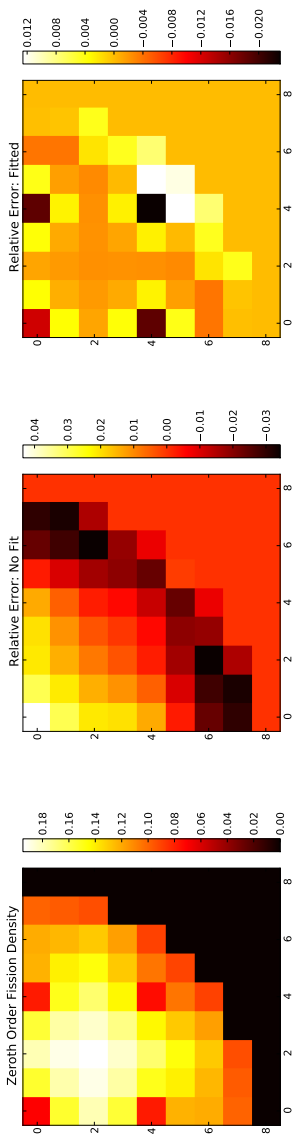


Figure 2.3: Reference (100x100 mesh) fission density with model relative errors.

2.6 LABAN-PEL

As noted above, time and other constraints made it infeasible to implement the zeroth order model into the intended code `serment`. As an alternative (and staying with the response function theme), the response matrix code LABAN-PEL was used for the studies in Chapter 4. The code is simple to use, and for explaining it here, an annotated input is provided. More details can be found in the user manual.

The input is for the Biblis benchmark, as described in the reference of Hebert. The annotations are lines beginning with # (which cannot be present when running the input). Only relevant parameters are pointed out. Note, the extra fission cross-section in the material definitions is related to generalized diffusion coefficient definitions, details of which can be found in the manual.

```
# Title.
 LABANPEL  UNRODDED BIBLIS TEST PROBLEM (HEBERT, NSE 91, 34 ,1985)  P-2, 1*1
# First four are numbers of groups, meshes along x, meshes along y, and materials.
    2    9    9    8    0    0    2  1.0
# First is Legendre expansion order.
    2    0    0    0    0    0    0    0     0
# 100 inners max, 10 outers max, and various convergence criteria of 0.0001
    1    0. 0.     0   100   10 .0001   .0001   .0001   1.0
    1    1    1    1    1    1    1    1    1
    1    1    1    1    1    1    1    1    1
# Mesh dimensions.
11.5613 23.1226 23.1226 23.1226 23.1226 23.1226 23.1226 23.1226 23.1226
11.5613 23.1226 23.1226 23.1226 23.1226 23.1226 23.1226 23.1226 23.1226
# Material map. -1 is reflection, -2 is vacuum, 0 is void.
   0  -1  -1  -1  -1  -1  -1  -1  -1  -1   0
  -1   1   8   2   6   1   7   1   4   3  -2
  -1   8   1   8   2   8   1   1   4   3  -2
  -1   2   8   1   8   2   7   1   4   3  -2
  -1   6   2   8   2   8   1   8   4   3  -2
  -1   1   8   2   8   2   5   4   3   3  -2
  -1   7   1   7   1   5   4   4   3  -2   0
```

```
  -1   1   1   1   8   4   4   3   3  -2   0
  -1   4   4   4   4   3   3   3  -2   0   0
  -1   3   3   3   3   3  -2  -2   0   0   0
   0  -2  -2  -2  -2  -2   0   0   0   0   0
#   D_g     Sigma_Ag    nuSigma_Fg chi_g  Sigma_12    nuSigma_Fg (again)
    1    5    1  MATERIAL 1
  1.4360   0.0095042   0.0058708   1.    0.       0.   0.0058708   1.
  0.3635   0.075058    0.096067    0.    0.017754 0.   0.096067    1.
    2    5    1  MATERIAL 2
  1.4366   0.0096785   0.0061908   1.    0.       0.   0.0061908   1.
  0.3636   0.078436    0.103580    0.    0.017621 0.   0.103580    1.
    3    5    1  MATERIAL 3 (REFLECTOR)
  1.3200   0.0026562   0.          1.    0.       0.   0.          1.
  0.2772   0.071596    0.          0.    0.023106 0.   0.          1.
    4    5    1  MATERIAL 4
  1.4389   0.010363    0.0074527   1.    0.       0.   0.0074527   1.
  0.3638   0.091408    0.132360    0.    0.017101 0.   0.132360    1.
    5    5    1  MATERIAL 5
  1.4381   0.010003    0.0061908   1.    0.       0.   0.0061908   1.
  0.3665   0.084828    0.103580    0.    0.017290 0.   0.103580    1.
    6    5    1  MATERIAL 6
  1.4385   0.010132    0.0064285   1.    0.       0.   0.0064285   1.
  0.3665   0.087314    0.109110    0.    0.017192 0.   0.109110    1.
    7    5    1  MATERIAL 7
  1.4389   0.010165    0.0061908   1.    0.       0.   0.0061908   1.
  0.3679   0.088024    0.103580    0.    0.017125 0.   0.103580    1.
    8    5    1  MATERIAL 8
  1.4393   0.010294    0.0064285   1.    0.       0.   0.0064285   1.
  0.3680   0.090510    0.109110    0.    0.017027 0.   0.109110    1.
# Designate that -1 and -2 are reflective and vacuum
WHITE
BLACK
```

CHAPTER 3

Heuristic Tie-Breaking Crossover

3.1 Genetic Algorithms

Genetic algorithms (GA) are a powerful class of algorithms for global optimization. The basis of GA's is the Darwinian theory of evolution, or in other words, survival of the fittest. GA's are inherently different from many other optimization techniques in that *populations* of solutions are moved in sequence rather than *individual* solutions. In fact, this basic trait is what makes GA's so easy to parallelize.

The basic GA is really quite straightforward and can be described in terms of a few simple "genetic" operators: *selection*, *crossover*, and *mutation* . A simple GA is given in Algorithm 1.

Fundamental to GA's is how strings are used to represent the solution space of interest. The original work on GA's used binary encoding, where sequences of 1's and 0's represented solutions. However, almost any type of representation can be used if the appropriate operators are defined (which will be studied below for core loading patterns). The rule-of-thumb is to use a representation that is natural for the problem [19]. This gives GA's

Create and **Evaluate** an initial population of strings
while *not converged* **do**
  Select strings for next population
  Crossover and/or **Mutate** strings
  Evaluate the population fitness
end

Algorithm 1: Simple GA Algorithm[26]

considerable flexibility when modeling complex systems.

The three genetic operators each play distinct but equally important roles in the optimization process. The crossover operator combines information from two parent strings and passes it to one (or more) offspring. Ideally, a crossover operator produces viable offspring for a natural representation of the solution space. For some problems, this can lead to relatively complex crossover operators (as will be discussed below). Probably the simplest crossover is a "one point crossover", for which a *crossover point* is chosen randomly that splits each parent into two partial strings. Two children are then produced using one portion of each parent. As an example, suppose a problem is represented in bit form and two parent solutions are `P1 = (0 1 0 0 1 1)` and `P2 = (1 1 0 1 1 0)`. If the crossover point is 2, two possible children are `C1 = (1 1|0 0 1 1)` and `C2 = (0 1|0 1 1 0)`.

To perform the crossover, two parents are needed from the current population. Typically, parents are selected for crossover based on their fitness—that is, how well they satisfy the objective. Many selection techniques have been proposed, but a common technique (and the PGAPack default) is "tournament selection". For each parent required, N members of the current population are selected randomly; the parent is the fittest of those selected. The number of children to be produced can also vary. In a "pure generational" replacement scheme, no member of the current population can live on (which, to some degree, mimics nature). A common

approach is to transfer some number of fittest strings of the current generation to the next, yielding various degrees of "elitism". The PGAPack default is to replace just 10 strings, a rather low number.

The crossover operator is typically the driving force in the beginning of the opimization sequence, but it is effectively a global operator and misses local improvements. Mutation accounts for this, and expands the search space by small, local perturbations. For binary strings, a simple "bit flip" is commonly used; mutation on `C2` above might yield `C2 = (0 1 0 1 1 1)`, where the last bit has been flipped. Generally, strings are subject to mutation only if they do not crossover (but that is not a rule).

The many varieties of GA implementations constitute a vast research topic. Here, only the basics have been summarized. Carter has provided a nice summary of GA's in the context of core loading pattern optimization [19]. The review article of Srinivas and Patnaik gives a nice overview of GA's. For implementation details related to this work, the PGAPack manual is the best resource [16].

3.2 Ordering Applications and The TSP

While GA's are flexible with respect to representing a solution space, there exists a class of problems called "ordering" or "permutation" problems for which a natural representation coupled with the standard genetic operators (*e.g.* bit-flipping mutation and n-point crossovers) does not readily apply. The canonical example is the *traveling salesman problem* (TSP). In the TSP, a salesman must visit some number of cities exactly once; the goal is to minimize the distance traveled. While trivial conceptually, the TSP becomes exponentially harder with increasing numbers of cities. Only for relatively small problems can solutions be found by direct approaches; for example, a 15112 city problem took 22.6 CPU years, a nontrivial cost, and "challenge" problems with millions of cities remain unsolved [27].

The difficulty arises in the crossover operation and less so for mutation. To illustrate, consider a problem for which a solution has six genes labeled alphabetically from `a` to `f`. In terms of the TSP, a potential solution (`f b d e c a`) indicates the salesman begins in `f`, travels to `b`, and so on. Figure 3.1 demonstrates the issue: a simple crossover can take two parents that are complete lists (*i.e.* has each element) and produce two offspring that are incomplete lists and hence invalid solutions. In this case, the first child has two `c` elements but no `f` elements and *vice versa* for the second child.

```
Parent 1:     (f b d e c a)
Parent 2:     (b d c e a f)

Standard "One Point Crossover"
Parent 1:    (f b d|e c a)
Parent 2:    (b d c|e a f)
Child 1:     (b d c e c a)
Child 2:     (f b d e a f)
```

Figure 3.1: One point crossover on a simple ordering problem.

Similarly, a standard mutation that selects a gene and assigns it a random value cannot be expected to yield valid results. For mutation, however, the easiest fix is to use a "swapping" mutation, where two elements in a solution are randomly selected and swapped. Of course, while this maintains a full list, it might otherwise introduce features in the solution that make it invalid, in which case a more sophisticated approach is necessary.

3.3 Tie-Breaking Crossover

To address the issues introduced by standard crossover of ordered lists, many new crossover operators have been proposed, including the Partially Mapped Crossover (PMX) [28], Order Crossover (OX) [29], Genetic Edge Recombination (GER) [30], and Tie-Breaking Crossover (TBX) [31] op-

erators, among many others. For this project, the focus is on the TBX operator, since it is the foundation for the related Heuristic Tie-Breaking Crossover operator used in this project as the primary crossover operator.

The TBX operator works by interpreting a parent string using the "position listing" of its elements. The position listing is an index that specifies where a possible gene is located in the parent string. For the first parent in Figure 3.1, the gene `a` is (alphabetically) the *first* possible gene but is the *sixth* gene of the parent. Hence, the first value of the position listing is six. Once the parent strings are represented via these position listings, normal crossover is applied. The novel feature of TBX is then to generate randomly a crossover map, which is applied to the parents as illustrate in Figure 3.2. The position listings are multiplied by the string size, and the map is added. The results are reordered, and the children are found by mapping back to the alphabetic genes. The nature of the crossover map application ensures the resulting children are valid.

3.4 Heuristic Tie-Breaking Crossover

In the TBX operator, the parent strings are recast in position listings where the position is based on some designation of individual genes. Were the six gene example above a TSP, each alphabetic gene represents some city, and the cities are assigned these alphabetic designations arbitrarily. For some problems, notably the core loading pattern problem, individual genes—assemblies with burnable poison specification, *etc.*— have much more associated with them than does a city in the TSP. For example, one likely knows an assembly's burnup or reactivity, among others things, and this information can be used in the listing process.

Noting that the listing need not be arbitrary, Poon and Carter introduced the Heuristic Tie-Breaking Crossover (HTBX) [32]. In this case, each gene is ranked by some criterion, and the listing procedure is based on

```
Parent 1:    (f b d e c a)
Parent 2:    (b d c e a f)

Position listing
Parent 1:    ( 6  2  5  3  4  1)
Parent 2:    ( 5  1  3  2  4  6)

"One Point Crossover" on listing
Parent 1:    ( 5  1  3| 3  4  1)
Parent 2:    ( 6  2  5| 2  4  6)

Random crossover map
Map:         ( 5  0  1  2  4  3)

Parents * 6 + map
Parent 1:    (35  6 19 20 28  9)
Parent 2:    (41 12 31 14 28  9)

Replace lowest by 1, then next by 2...
Parent 1:    ( 6  1  3  4  5  2)
Parent 2:    ( 6  2  5  3  4  1)

Map numbers back to characters
Child 1:     (b f c d e a)
Child 2:     (f b d e c a)
```

Figure 3.2: TBX on a simple ordering problem.

this ranking. Otherwise, HTBX is identical to TBX. Figure 3.3 illustrates the idea for the same six gene problem, where each gene `a` to `f` is ranked in some non-alphabetical way.

Ultimately, the effect of using the ranking scheme is to produce offspring that are similar to the parents with respect to whatever is used to determine the rankings. For the case of core loading pattern problems, the ranking can be based on reactivity, quantified by k_∞, so that offspring are produced that preserve gross neutronic features of the parents better on average than would an arbitrary listing scheme. This is potentially beneficial, since extreme perturbations to a given pattern often lead to equally extreme changes in the objective.

3.5 Numerical Examples

To illustrate the TBX and HTBX operators, two simple but meaningful problems are considered. All the numerical results are simply for representative cases. Averages over several runs would be needed to make any rigorous conclusions.

Oliver's 30 City TSP

The first is Olivers's 30 city TSP [29], the coordinates for which can be found online*. The problem has 40 equivalent optima, each with a distance of 423.741 units. This problem has been solved by a number of methods, and a good summary of results can be found in Ref. [31].

Here, the problem is solved by TBX using 500 generations, a population size of 50, crossover probability of 85%, and a swap mutation rate of 5%. Only 10 strings (of the 50) are replaced, which is the PGAPack default. For this set of parameters, 2415 evaluations were required. For comparison, a random search was also employed using 2415 evaluations. The results for both TBX and the random search are presented in Figure 3.4 for a representative case. For the random search, the x-axis has been scaled using 2415/500 evaluations per generation. As one can see, the GA with TBX does much better than the random search as can be expected. Considering the search space is large—$30! \approx 3^{32}$—the GA does quite well. Note, this problem with some slightly different parameters is contained in Examples 5 and 9 of `pypgapack` [18].

Slab Reactor

As a second problem, we consider a simple one dimensional slab reactor using two group diffusion theory. The reactor consists of 10 slabs, each 20 cm

*See `http://www.stevedower.id.au/other/oliver30`

in width. The first and last slabs are reflectors, and the central 8 slabs are fuel. The fuel materials represent a particular assembly with increasing burnup from 0 through 35 MWd/kg burnup in 5 MWd/kg increments. That is, fuel material 0 has 0 burnup, while fuel material 7 has 35 MWd/kg. The left and right boundary conditions are reflective and vacuum, respectively, so that mirror image patterns are not identical. The problem is relatively small, with only $8! = 40320$ possible patterns, and so the entire search space can be computed directly. One version of this is contained in Example 10 of of `pypgapack` [18].

Our interest here is in the eigenvalue k and the slab power peaking factor p, defined as the ratio of the maximum average power in a slab to the average power in the reactor. Figure 3.5 shows the tradeoff between the eigenvalue and p for all possible combinations.

The exact objective to minimize is a key decision. In general, aiming for a larger eigenvalue causes the p to increase and *vice versa*. Hence, the problem is inherently a multiobjective one, and much work has investigated how to treat the multiple objectives somewhat directly; see for example Ref. [33].

For this project, a multiobjective treatment is outside the intended scope (though should be a topic of future efforts, as adding multiobjective capabilities in `pypgapack` should be straightforward). Instead, a single weighted objective function is to be studied. Following Ref. [8], we seek to maximize an objective of the form

$$f(p,k) = w_p \delta_p + w_k (k - 1.0)\,. \tag{3.1}$$

where

$$\delta_p = \begin{cases} p - p_{\max} & \text{if } p - p_{\max} > 0, \\ 0 & \text{otherwise}\,. \end{cases} \tag{3.2}$$

We consider two cases. For the first case, $p_{\max} = 1.8$, $w_p = -10$, and $w_k = 1$. For the second case, $p_{\max} = 1.5$, $w_p = -1$ and $w_k = 2$. The reference solutions are given in Table 3.1.

	Case 1	Case 2
$f(p,k)$	1.06611227	2.07254835
k	1.06611227	1.03627417
p	1.75864379	1.49120332

Table 3.1: Reference solutions for slab reactor.

To solve the problem, HTBX is used with ranking based on k_∞. 300 generations are used. All other parameters have the same values as used in the TSP problem above. Actually, by construction, HTBX is equivalent in this case to TBX since the materials are listed *a priori* by decreasing k_∞ (as a result of increasing burnup). In fact, this idea should apply to all problems, as ranking must be based on information known before evaluating the pattern. Hence, all possible fuel assemblies can be presorted by reactivity (or some other feature) so that the ranking is inherently represented by the numerical index of the assembly in solution space. The approach makes HTBX more efficient, as it avoids sorting during every crossover operation. It is assumed this is what has been done in practical implementations, but none of the references actually mention it.

Figure 3.6 shows the results for Case 1. The solution found was $f(k,p) = k = 1.06610443$ and $p = 1.76210154$, very close to the reference solution. In fact, the reference value was the only one better. Worth noting is the actual pattern, `(6 2 1 5 3 0 4 7)`. The reference case is `(6 2 0 5 3 1 4 7)`. Hence the only difference is the placement of the two materials with lowest burnup. Note, 1491 evaluations were used.

Figure 3.7 shows the results for Case 2. The solution found was $f(k,p) = 2.07238023$, $k = 1.03619012$, and $p = 1.49226572$, very close to the reference solution. This time, three cases were better. The pattern is `(0 1 7 3 6 4 2 5)`, and the reference is `(1 0 7 3 6 4 2 5)`. Again, the only difference is the placement of the two materials with lowest burnup. Note,

1452 evaluations were used.

```
Parent 1:    (f b d e c a)
Parent 2:    (b d c e a f)

Rank Genes
(a b c d e f) → (2 4 1 5 6 3)

Position listing
Parent 1:    ( 3  4  5  6  1  2)
Parent 2:    ( 4  5  1  6  2  3)

"One Point Crossover" on listing
Parent 1:    ( 4  5  1| 6  1  2)
Parent 2:    ( 3  4  5| 6  2  3)

Random crossover map
Map:         ( 5  0  1  2  4  3)

Parents * 6 + map
Parent 1:    (29 30  7 28 10 15)
Parent 2:    (23 24 31 28 16 21)

Replace lowest by 1, then next by 2...
Parent 1:    ( 5  6  1  4  2  3)
Parent 2:    ( 3  4  6  5  1  2)

Map numbers back to characters
Child 1:     (d e c b a f)
Child 2:     (f b e d c a)
```

Figure 3.3: HTBX on a simple ordering problem.

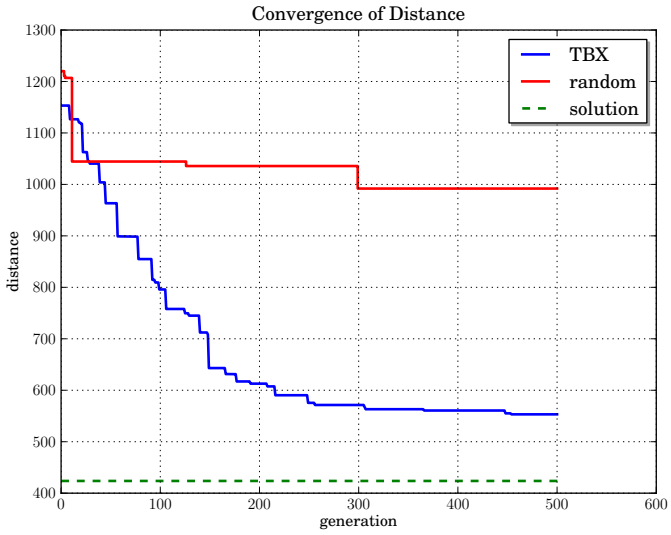


Figure 3.4: TBX versus random search for Oliver's 30 city TSP.

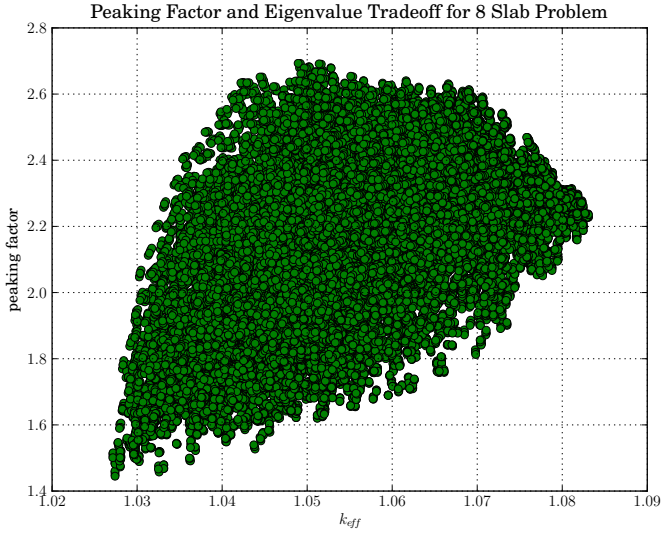


Figure 3.5: Tradeoff between the eigenvalue and peaking factor.

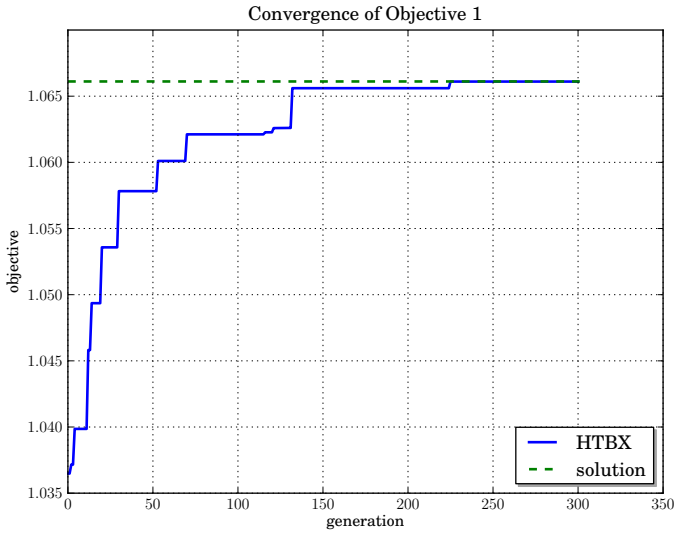


Figure 3.6: Case 1 convergence.

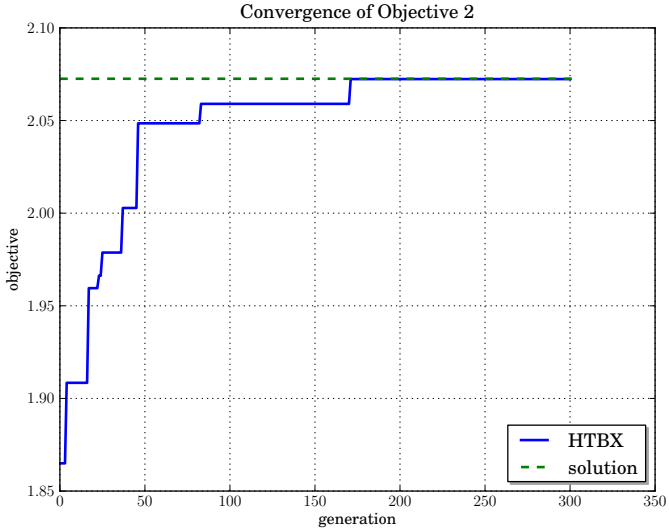


Figure 3.7: Case 2 convergence.

CHAPTER 4

poropy: A Simple Tool

Using the tools described in the previous chapters, this chapter demonstrates a simple tool poropy that can be used to study cores manually and optimize patterns automatically. A brief overview is given of the current structure, and shows the patterns put in place for extensions. Thereafter, an example is used demonstrate the tool.

4.1 Overview

The core of poropy has already been discussed: namely the neutronics (here handled by LABAN-PEL) of Chapter 2 and the GA and HTBX operator of Chapter 3. Here, those tools are pulled together in a common Python interface. The tool is currently rather sparse, as more work was spent testing the other components. However, the basic structure is solid enough for future additions.

poropy consists of two packages: coretools and bundletools. The former is the focus here, while the latter will be, when finished, a cleaned up version of some previous homeworks involving bundle optimization.

The `coretools` package has several modules, the important ones being `optimizer`, `reactor`, `assembly`, `vendor`, and `laban`, each which has a class of the same name (*i.e.* reactor has the `Reactor` class). The `vendor` module is not implemented yet.

The `Reactor` class contains in some sense everything needed to optimize a core loading pattern. It contains a list of `Assembly` objects that are currently in the core, and will in the future have another list of `Assembly` objects that are in the spent fuel pool. The class `Vendor` will provide definitions for different fresh assemblies that can be inserted. The `Reactor` object is defined using a "stencil", indicating *where* fuel can go, and a "pattern", indicating *what* fuel goes in each fuel location. Fuel "regions" can be specified. For now, this is an unused feature but could be used to define regions of fixed fuel (such as the central bundle or bundles along a symmetry line). For the unfueled regions, `Reflector` or void can be designated. The reflector material is defaulted to the Biblis reflector, but the user can change this. Currently, the modeling is limited to quarter core geometry with rotational symmetry (even though LABAN-PEL does not have rotational boundary conditions). It is assumed use of the "stencil" would make generalizations straightforward.

The `Reactor` class also has functions for changing and evaluating the loading pattern. A user can swap bundles via the `Reactor.swap([x_1,y_1],[x_2,y_2])` command, where `[x_i,y_i]` defines a location within the stencil. Future additions will facilitate fresh fuel exchanges via the `Vendor`, either individually or for the entire fresh fuel inventory.

Once a swap has been made, the user can evaluate the pattern using `Reactor.evaluate()`, and then print the parameters of interest via `Reactor.print_params()`; currently, only the eigenvalue and peaking factor are given. The pattern can be plotted by `Reactor.plot_pattern()`, and the peaking can be plotted by `Reactor.plot_peaking()`.

`Reactor.print_pattern()` and `Reactor.print_peaking()` do the same using text output. Similar features for the group fluxes would be easily added.

To evaluate patterns, the user must create an evaluator. The only option currently is the `Laban` class. The current structure of `Laban` could be abstracted out to form an `Evaluator` base class in the future. The interface between `Reactor` and `Laban` is relatively generic. Essentially, the core information (which actually resides in an object of class `Core`) is passed to `Laban` for input generation and output parsing. That same information could be similarly used by any evaluator.

Each `Assembly` object contains all the information associated with the assembly. Currently, only group constants are stored, but burnup, orientation, and other data could be kept for future additions. The interface is designed so that the `Assembly` object gives its data when requested. In this way, plugging in a more sophisticated data model (as developed by another group) should be trivial. Currently, each `Assembly` object must be assigned material definitions; again, with a data model, this could be an automated process based on a few parameters.

Finally, the `Optimizer` class (derived from the `PGA` class of `poropy`) can be used to optimize a given `Reactor` object. Here, the coupling remains rather loose, as it seems better for now to keep the optimizing object separate. However, the `Reactor` class methods are used in the evaluation.

This description has been intentionally brief. The Sphinx-generated documentation is more complete, and in particular, the examples (results of which are given below) are the best way to see the syntax in action. The documentation and code will be moved online as soon as built-in neutronics module is completed.

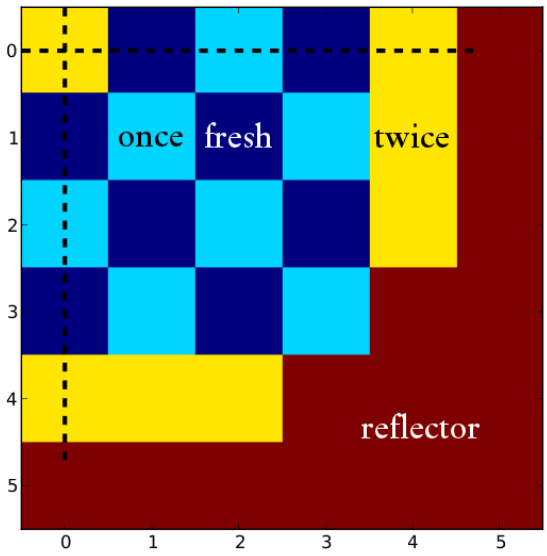


Figure 4.1: Small three fuel material core.

4.2 A Simple Benchmark

As in Chapter 3, the efficacy of the optimizer is best illustrated using a problem for which a solution is known. Here, we define a simple small core consisting of 69 assemblies and three materials representing fresh, once burned, and twice burned fuel plus a reflector. These materials are the same as the checkerboard test case of Chapter 2. All assemblies are 23.1226 in width. An initial core pattern is given in Figure 4.1. The dark blue regions are fresh fuel, the light blue once burned, and yellow twice burned. Red is reflector, here taken from the Biblis benchmark.

Solution Space

This pattern was chosen because the entire search space can be computed directly. While the boundaries are reflective in modeling, rotational sym-

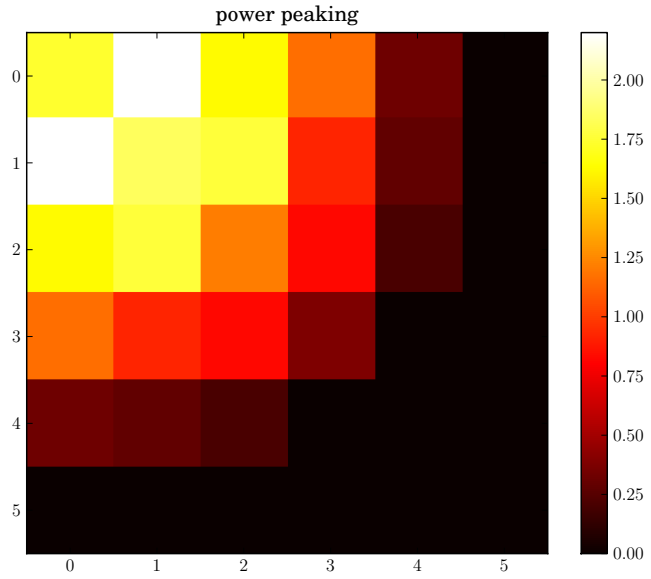


Figure 4.2: Small core peaking for the pattern of Figure 4.1.

metry is enforced in placement. Hence, there 6 unique locations for each fuel type. However, the central assembly is forced to be the twice burned fuel. This leaves just 17 unique locations, or

$$\text{number of patterns} = \binom{17}{6} \times \binom{11}{6} = \frac{17!}{6!6!5!} = 5,717,712\,. \tag{4.1}$$

It is quite astounding that such a seemingly small problem can create such a large solution space.

For the initial pattern, $p = 2.20561$ and $k = 1.157522$ using second order expansions. These are 2.20006/1.157152 and 2.0687/1.137119 for first and zeroth order, respectively. For the sake of efficiency, zeroth order is used. For reference, the pattern of Figure 4.1 yields the peaking pattern given in Figure 4.2.

To compute all patterns, MATLAB's useful combinatorial functions were used to generate a text file with each pattern. Python probably has

similar functions, but this was not investigated. The entire set of patterns was evaluated using 4 cores in a little over 7 hours of wall time, which is about the 20 million per day. One thing that became apparent is that no matter what the speed of the underlying neutronics kernel is, Python will add nontrivial overhead. For this case, the input writing and output parsing is expected to be relatively expensive as overheads go; the average per-pattern time ranged from 0.01 to 0.02 s on a single core when the actual computations (in LABAN-PEL) require roughly 0.005 s. It is unclear how much overhead can be eliminated, but it might mean putting more of the routines in C/C++ will be needed to reach the 200 million per day goal on single multicore machines.

Figure 4.3 shows the peaking factor against the eigenvalue for all cases*. The overall shape is actually quite similar to that of the slab problem in Figure 3.5, which suggests the simpler slab problem might be more representative than first imagined. Some unique, nearly banded structures appear for large k and p, with one band appearing to nearly completely separated. Other unique features include a few well-separated solutions along the lower surface of the main body of solutions. For simple objectives based on k and p alone, a curve bounding this lower surface of the scatter plot would essentially quantify everything needed. In other words, for a given k, the curve would give approximately the lowest p possible and *vice versa*†.

A Manual Study

Before moving to optimization, it is worth probing the problem by hand. This helps illustrate a bit of the syntax of `poropy` as well. The demon-

*The image quality is worse because the PDF image straight from Python was over 40 MB. LaTeX didn't approve, so a screen shot was used

†Ultimately, it is exactly that sort of bounding tradeoff curve that optimization seeks to explore. In weighted objective schemes, such a curve is found point-by-point, while true multiobjective treatments can find the curve directly during one optimization.

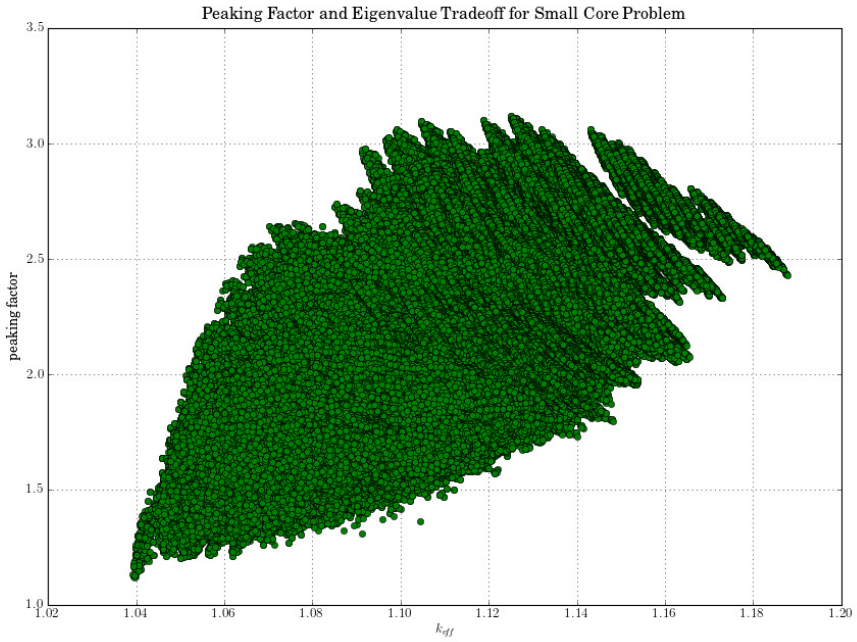


Figure 4.3: Tradeoff between the eigenvalue and peaking factor.

stration largely following the Example 1 for the small core in the `poropy` documentation.

The small core problem is produced by a help script, which is neglected here for brevity. However, we can load that and define the small reactor object:

```
import small_core
reactor = small_core.make_small_core()
```

To ensure that it is properly constructed, we can display the contents of everything via

```
reactor.display()
```

The output is rather lengthy, as each individual fuel assembly is displayed. A portion of that output is

```
 poropy - diagnostic output
```

```
REACTOR:
    thermal power : 1000 MWth
   electric power : 1000 MWel
    CORE:
       pattern:    [12  0  6  1 15  8  2  9 13  4 10  5 16 11  3  7 14 17]
        ASSEMBLIES:
            assembly:  0
                model:  IFBA
           enrichment:  4.25
               burnup:  0.0
         constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
                kinf:  1.29677543186
              assembly:  1
                 model:  IFBA
            enrichment:  4.25
                burnup:  0.0
         constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
                kinf:  1.29677543186
           ...
              assembly:  17
                 model:  IFBA
            enrichment:  4.25
                burnup:  30.0
         constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
                kinf:  0.994529582556
        REFLECTOR:
                model:  biblis
        constants:  1.32 0.2772 0.0026562 0.071596 0.0 0.0 0.023106 0.0257622
    EVALUATOR:  LABAN-PEL
        input:  laban0.inp
       output:  laban0.out
```

Evaluating the default pattern is easy via

```
reactor.evaluate()
reactor.print_params()
reactor.print_peaking()
```

which yields the output

```
 optimization parameters
 -----------------------
    keff   =  1.137119
 maxpeak   =  2.0687

[[ 1.64865  2.0687   1.53543  1.18246  0.40744  0.     ]
 [ 2.0687   1.7359   1.69436  0.94983  0.34894  0.     ]
 [ 1.53543  1.69436  1.19476  0.8788   0.26544  0.     ]
 [ 1.18246  0.94983  0.8788   0.43841  0.       0.     ]
 [ 0.40744  0.34894  0.26544  0.       0.       0.     ]
 [ 0.       0.       0.       0.       0.       0.     ]]
```

Manually swapping bundles is also straightforward. The bundles are indexed by (i, j), beginning with zero. First, we print the pattern:

```
reactor.print_pattern()
```

yielding

```
 current loading pattern
 -----------------------
       0   1   2   3   4
  ----------------------
 0|   12   6   0   1  15
 1|   rs   8   2   9  13
 2|   rs   4  10   5  16
 3|   rs  11   3   7 ref
 4|   rs  14  17 ref ref
```

The "rs" and "ref" indicate rotational symmetry and reflector, respectively. Looking at this and the previous peaking, a potentially good swap to reduce peaking is $(0, 1)$ with $(0, 2)$ (or bundle 6 with bundle 0). Note that the bundles are ranked by reactivity. Hence, bundles 12 and up are the least reactive, and so on. This is done to avoid excessive sorting in the GA (as noted above). Making the swap,

```
reactor.swap([0,1],[0,2])
```

```
reactor.print_pattern()
reactor.evaluate()
reactor.print_params()
reactor.print_peaking()
```

we get the output

```
current loading pattern
 ------------------------
       0   1   2   3   4
  ----------------------
 0|   12   6   0   1  15
 1|   rs   8   2   9  13
 2|   rs   4  10   5  16
 3|   rs  11   3   7 ref
 4|   rs  14  17 ref ref


 optimization parameters
 ------------------------
    keff   =  1.135264
 maxpeak   =  1.85174

[[ 1.22885  1.47989  1.85174  1.31604  0.4458   0.     ]
 [ 1.47989  1.53051  1.73407  1.0157   0.37678  0.     ]
 [ 1.85174  1.73407  1.21558  0.91612  0.2809   0.     ]
 [ 1.31604  1.0157   0.91612  0.45609  0.       0.     ]
 [ 0.4458   0.37678  0.2809   0.       0.       0.     ]
 [ 0.       0.       0.       0.       0.       0.     ]]
```

Hence, the peaking is reduced a bit without much change in k. Better patterns are available for this k. Figure 4.3 suggests that for $k \approx 1.135$ there is a peaking around 1.75.

An Optimization Study

To illustrate the optimization capability of `poropy`, we continue where the manual study ended: find the smallest p such that $k \geq 1.135$. In a form similar to Eq. 3.1, our goal is to maximize

$$f(p,k) = w_p(1.85 - p) + w_k\delta_k \,, \tag{4.2}$$

where

$$\delta_k = \begin{cases} k - 1.135 & \text{if } k - 1.135 < 0, \\ 0 & \text{otherwise .} \end{cases} \tag{4.3}$$

For this example, we use $w_p = 1$ and $w_k = 50$; the relatively high weight for k should guarantee the "hard" constraint is met.

The reference solution for this was found to be $f = 0.11749$, $k = 1.14012$, and $p = 1.73251$. As expected, the optimum occurs for a pattern satisfying the k constraint. For the optimization run, the same parameters as the slab problem of Chapter 3 were used, except 500 generations were used. In one case, we use the default PGAPack replacement of 10 strings; that is, every generation, just 10 patterns are replaced. For a second case, 40 strings are replaced, meaning the best 10 are kept. A summary of results is given in Table 4.1. For both cases, 10 runs with different random number generator seeds were used, but they were the same seeds for each case. The standard deviations are included. As can be seen, for Case 2, the

	Reference	Case 1	Case 2
$f(p,k)$	0.11749	0.11125 ± 0.01871	0.11749 ± 0.0
k	1.14012	1.13979 ± 0.00099	1.14012 ± 0.0
p	1.73251	1.73875 ± 0.01871	1.73251 ± 0.0
evaluations	n/a	5060	20090

Table 4.1: Solutions for small core reactor.

optimum value is achieved, while for just one quarter of the evaluations, Case 1 offers an average value quite close to the reference; in fact, only one run of 10 failed to find the optimum. For reference, the optimum loading pattern is

```
current loading pattern
-----------------------
      0   1   2   3   4
 ----------------------
0|   17   7  10   5   6
1|   rs   9   2   0  14
2|   rs   4   1  11  13
3|   rs   3   8  16 ref
4|   rs  12  15 ref ref
```

and converting this back to the original indices of 0, 1, and 2 for fresh, once-, and twice-burned, we have

```
current loading pattern
-----------------------
      0   1   2   3   4
 ----------------------
0|    2   1   1   0   1
1|   rs   1   0   0   2
2|   rs   0   0   1   2
3|   rs   0   1   2 ref
4|   rs   2   2 ref ref
```

This is quite symmetric and is similar to "ring of fire" type cores. The associated power peaking distribution is

```
[[ 1.07658  1.28113  1.4042   1.37252  0.58032  0.     ]
 [ 1.28113  1.4031   1.73251  1.30587  0.43526  0.     ]
```

```
[ 1.4042   1.73251  1.53302  0.78678  0.27084  0.     ]
[ 1.37252  1.30587  0.78678  0.34406  0.       0.     ]
[ 0.58032  0.43526  0.27084  0.       0.       0.     ]
[ 0.       0.       0.       0.       0.       0.     ]]
```

In general, the results suggest that higher replacement leads to a better solution, but at a higher function evaluation count. For the Case 2, all 10 runs achieved the same (optimum) solution and hence had a standard deviation of zero. The tradeoff between function evaluations and convergence is always problem dependent, but in general "elitism" has worked well in GA's. This implies that the portion kept from generation to generation is relatively small, as is true for Case 2.

The objective above was relatively easy to solve exactly—a suprising feat given the solution space is still orders of magnitude larger than the number of evaluations used. A slightly more challenging problem might be to define an objective whose solution is one of the well-separated points in Figure 4.3. For this, we'll seek the minimum p for $k < 1.1$, whose solution is the obvious low point just right of $k = 1.10$. Similar to the objective above, we maximize

$$f(p,k) = w_p(1.50 - p) + w_k\delta_k\,, \tag{4.4}$$

where

$$\delta_k = \begin{cases} k - 1.11 & \text{if } k - 1.11 < 0, \\ 0 & \text{otherwise}\,, \end{cases} \tag{4.5}$$

and where again $w_p = 1$ and $w_k = 50$. Cases 1 and 2 were rerun with the new objective, and the results are given in Table 4.2.

For this objective, both Case 1 and Case 2 have a much harder time converging (though for Case 1, one of the ten runs was optimal, and for Case 2, two were optimal). Overall, Case 2 showed better performance, which can be expected simply by the greater number of evaluations used. For reference, the optimal pattern is

	Reference	Case 1	Case 2
$f(p,k)$	0.13595	0.02905 ± 0.03866	0.04307 ± 0.04885
k	1.10442	1.10371 ± 0.00280	1.10396 ± 0.00186
p	1.36405	1.47095 ± 0.03866	1.45693 ± 0.04885
evaluations	n/a	5060	20090

Table 4.2: Solutions for small core reactor for second objective.

```
current loading pattern
-----------------------
      0   1   2   3   4
 ---------------------
0|    2   1   1   0   2
1|   rs   1   1   0   2
2|   rs   1   1   0   2
3|   rs   0   0   0 ref
4|   rs   2   2 ref ref
```

and the associated power peaking distribution is

```
[[ 1.01074  1.17609  1.25641  1.36405  0.53696  0.     ]
 [ 1.17609  1.20045  1.25763  1.36116  0.5025   0.     ]
 [ 1.25641  1.25763  1.23066  1.19863  0.39572  0.     ]
 [ 1.36405  1.36116  1.19863  0.80143  0.       0.     ]
 [ 0.53696  0.5025   0.39572  0.       0.       0.     ]
 [ 0.       0.       0.       0.       0.       0.     ]]
```

This pattern brings the fresh fuel closer to the periphery, and in corner locations, gives rise to more leakage (which could explain the k difference between this and the previous objectives). However, by bringing the fresh fuel outward, the peaking is significantly reduced.

CHAPTER 5

Conclusion

5.1 Summary

This project set out with the task of creating a simple tool for core loading pattern optimization. Two major subtasks originally defined were to *create a fast neutronics kernel* and to *employ use of a genetic algorithm library*.

For the the neutronics kernel, a semi-analytic, zeroth order response matrix method was developed. The model was improved by a parameterizing the response function forms and fitting the result to reference fine mesh-generated response data. For an example core consistent with the reference data used to generate the fits, the unfitted model is accurate to within about 3% and the fitted model to within about 0.2% for maximum relative fission density errors. Those errors increase to about 4% and 2% for a core inconsistent with the reference data. Despite the promise of the method, time and technical constraints prevented implementation for the optimization analysis of the work. Instead, the code LABAN-PEL was used.

For the optimization, the GA library PGAPack was wrapped in Python

to create `pypgapack`, which was then used to create `poropy`. The basic capabilities `pypgapack` were demonstrated in Chapter 3 for two simple problems, and the additional features in `poropy` were demonstrated for a simple but reasonable core loading pattern problem in Chapter 4.

5.2 Suggested Future Work

A key takeaway from this project is that it is not impossible to demonstrate relatively high level optimization techniques for loading pattern optimization within a fairly small amount of code (and for problems that are somewhat meaningful). Because a basic framework has been put in place, there are a number of interesting areas that can be explored.

Heuristics

First, the literature is full of work on heuristics for optimization; examples relevant to core loading include Refs. [34, 35, 36, 37, 38]. Heuristics are useful as an optimization technique directly or, in the case of GA's, as a hill-climbing function used after each generation or as rules to be met before evaluation. While they move the problem away from one of pure optimization, "heuristics direct a time-limited optimization to patterns consistent with engineering judgement" [37]. No heuristics beyond the fixed central bundle have been built into `poropy` at this point due to time considerations, but several relational heuristics (*e.g.* no adjacent fresh bundles) would be easy to implement.

Other Optimization Schemes

Also, this project has focused solely on use of genetic algorithms. Other optimization schemes, particularly parallel simulated annealing, have been used successfully [3]. Ultimately, the tool would be most useful if a variety

of optimization techniques were implemented, giving users (especially students) a platform for learning about optimization methods. A brief search suggests a parallel version of the algorithm used in Ref. [3] is freely available, though the implementation is not at the same production level as PGAPack. Other approaches to consider are more active heuristics, including the *greedy binary sweep*, in which each bundle is swapped with every other bundle, and the swap is accepted if it improves the objective. The sweeps continue until a maximum number is reached or the solution stagnates. The similar *greedy dual binary sweep* extends the idea by using two swaps at once, which roughly corresponds to the maximum perturbation a human optimizer can hope to understand and exploit. Both of these could also be used with GA as hill-climbers. Other possibilities are branch and bound algorithms and linearized approaches that employ the simplex method repeatedly.

Neutronics

Additionally, while the neutronics efforts described in 2 are not implemented, it is still believed such an approach would offer a fast solver. The zeroth order approximation in general is not very accurate, and achieves maximum relative errors in the fission density up to 40%; this is still better than a normal coarse mesh finite difference approach, which yields errors of 50 or 60%. Going to first order reduces this to a few percent for common benchmarks [13]. Some initial work to extend the zeroth order fits to first order suggests just one odd transverse function could be enough to provide a reasonable model for use in a nonlinear regression fit without adding much to the already small cost. The current plan is to implement a response matrix solver using these fits "by hand" in Fortran, relying only on basic BLAS routines where necessary (and possibly the ARPACK library for its Arnoldi solver). The idea is that the problems of interest here will fall into a very narrow range that can be implemented without all the gener-

alities. Such a cut-and-dry implementation should be easy to analyze with tools like TAU (general profiling) and Papi (instrumentation for counting FLOPS and cache misses). The outcome should be *very* fast evaluations.

For the evaluations, any nodal method could similarly be used, though the more historical methods like FLARE would be fastest. A really valuable tool only somewhat related to this project (but definitely to another project) would be to implement as many of the various nodal and other coarse mesh methods as possible in one code. These methods are described in the literature, even more so in technical reports (it seems), but having them actually implemented would be a good way to maintain the knowledge, as it were. The approach in ROSA is some sort of one-and-a-half group approach that, given the description in Ref. [39], sounds like a collision probability method of sorts. It might be worth a deeper look.

Implementation

With respect to implementation, `poropy` currently operates via scripts or an interactive Python session, and there are lots of practical features to be added. While the scripts are fine for optimization runs, the interactive use is not ideal. Libraries like PyQt make GUI creation pretty straightforward. Given the back end is already in place, adding a basic GUI would take little time for a Python programmer with GUI experience. Moreover, it was noted before that Python adds nontrivial overhead. The most substantial overhead in the studies was largely due to the file reading and writing in wrapping LABAN-PEL. This would be eliminated by a direct interface. Additionally, several routines on the Python do some computational work. The most significant during optimization is the HTBX crossover operator. It was kept in Python for simplicity, but it could be implemented in C and interfaced to Python via SWIG. The same is true for just about all other routines.

Interdisciplinary Work

Finally, several students noted that their experience with XIMAGE was almost like playing a video game; one tries things, develops a sense of what works and what doesn't, and then applies that knowledge incrementally until the whole process seems (perhaps) easier and the level (or core) is won. Pertinent questions regarding that experience include "does everyone gain the same 'knowledge'?" and "do the 'tricks' used successfully have a common theme?" An interesting study would be to release something like `poropy` (with a GUI and a working full cycle framework) as a "game" for download or webplay. The goal would be to codify in some manner the approaches taken to get various scores and develop some sort of giant heuristic model—which sounds a bit like neural networks, though with a rather unique training scheme. At any rate, understanding the physics goes a long way in developing good patterns, but could gamers tell us more? Of course, a study like this doesn't belong (solely) in NSE—perhaps groups in CSAIL make sense. While way beyond the scope of this or likely any other class project, it's a neat idea that might be great for the many grants favoring interdisciplinary efforts.*

*Upon revision, the author was informed by his wife that a very similar idea has been put to use, described in a recent NPR segment[40] and published more recently in Science[41]. The work uses a game called `foldit` to allow players to "fold proteins". The game and more information is at their site: `http://fold.it/portal/info/science`. Despite being significantly less novel than originally thought, the idea noted above remains an interesting one.

Bibliography

[1] P. Turinsky, G. Parks, Advances in Nuclear Guel Management for Light Water Reactors, Advances in Nuclear Science and Technology (2002) 137–165.

[2] M. Driscoll, T. Downar, E. Pilat, The Linear Reactivity Model for Nuclear Fuel Management, American Nuclear Society, 1990.

[3] D. Kropaczek, COPERNICUS: A Multi-Cycle Optimization Code for Nuclear Fuel Based on Parallel Simulated Annealing with Mixing of States, Progress in Nuclear Energy 53 (6) (2011) 554–561.

[4] J. Stevens, K. Rempe, Equilibrium Cycle Analysis with XIMAGE/SIMAN, Proc. of the ANS Topical Meeting–Advances in Nuclear Fuel Management III.

[5] F. Verhagen, P. Walker, ROSA, a Flexible Loading Pattern Optimization Tool for PWRs, in: Proc. of the ANS Topical Meeting–Advances in Nuclear Fuel Management III, 2003.

[6] D. Kropaczek, P. Turinsky, In-core Nuclear Fuel Management Optimization for Pressurized Water Reactors Utilizing Simulated Annealing, Nuclear technology 95 (1) (1991) 9–32.

[7] B. Moore, P. Turinsky, A. Karve, FORMOSA-B: A Boiling Water Reactor In-Core Fuel Management Optimization Package, Nuclear Technology 126 (2).

[8] M. DeChaine, M. Feltus, Comparison of Genetic Algorithm Methods for Fuel Management Optimization, in: Proceedings of the International Conference on Mathematics and Computations, Reactor Physics, and Environmental Analyses, Vol. 1.

[9] J. Francois, H. Lopez, SOPRAG: A System for Boiling Water Reactors Reload Pattern Optimization Using Genetic Algorithms, Annals of Nuclear Energy 26 (12) (1999) 1053–1063.

[10] J. Zhao, B. Knight, E. Nissan, A. Soper, FUELGEN: A Genetic Algorithm-based System for Fuel Loading Pattern Design in Nuclear Power Reactors, Expert Systems with Applications 14 (4) (1998) 461–470.

[11] A. Erdoğan, M. Geçkinli, A PWR Reload Optimisation Code (XCore) Using Artificial Neural Networks and Genetic Algorithms, Annals of Nuclear Energy 30 (1) (2003) 35–53.

[12] F. Alim, K. Ivanov, S. Levine, New Genetic Algorithms (GA) to Optimize PWR Reactors: Part I: Loading Pattern and Burnable Poison Placement Optimization Techniques for PWRs, Annals of Nuclear Energy 35 (1) (2008) 93–112.

[13] E. Muller, Z. Weiss, Benchmarking with the Multigroup Diffusion High-Order Response Matrix Method, Annals of Nuclear Energy 18 (9) (1991) 535–544.

[14] S.-O. Lindahl, Multi-Dimensional Response Matrix Method, Ph.D. thesis (1976).

[15] E. Müller, LABAN-PEL: A two-dimensional, multigroup diffusion, high-order response matrix code, Tech. Rep. PEL 309 (June 1991).

[16] D. Levine, Users Guide to the PGAPack Parallel Genetic Algorithm Library, Argonne National Laboratory 9700.

[17] J. Roberts, pypgapack: Github repository (Dec. 2011).
URL `https://github.com/robertsj/pypgapack`

[18] J. Roberts, pypgapack: Online documentation (Dec. 2011).
URL `https://robertsj.github.com/pypgapack`

[19] J. Carter, Genetic Algorithms for Incore Fuel Management and Other Recent Developments in Optimisation, Advances in Nuclear Science and Technology (2002) 113–154.

[20] R. Lawrence, Progress in Nodal Methods for the Solution of the Neutron Diffusion and Transport Equations, Progress in Nuclear Energy 17 (3) (1986) 271–301.

[21] N. Gupta, Nodal Methods for Three-Dimensional Simulators, Progress in Nuclear Energy 7 (3) (1981) 127–149.

[22] A. Shimizu, K. Monta, T. Miyamoto, Application of the Response Matrix Method to Criticality Calculations of One-Dimensional Reactors, Nippon Genshiryoku Gakkaishi (Japan) 5.

[23] S. Lindahl, F. Weiss, The Response Matrix Method, Adv. Nucl. Sci. Technol. 13.

[24] A. Hébert, A Simplified Presentation of the Multigroup Analytic Nodal Method in 2-D Cartesian Geometry, Annals of Nuclear Energy 35 (11) (2008) 2142–2149.

[25] P. Pesic, Abel's Proof, MIT press, 2003.

[26] M. Srinivas, L. Patnaik, Genetic algorithms: A survey, Computer 27 (6) (1994) 17–26.

[27] J. Schneider, S. Kirkpatrick, Stochastic Optimization, Springer Verlag, 2006.

[28] D. Goldberg, R. Lingle, Alleles, Loci and the Traveling Salesman Problem, in: Proceedings of the First International Conference on Genetic Algorithms and Their Applications, 1985, pp. 154–159.

[29] I. Oliver, D. Smith, J. Holland, A Study of Permutation Crossover Operators on the Traveling Salesman Problem, in: Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application, 1987, pp. 224–230.

[30] L. Whitley, T. Starkweather, D. Fuquay, Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator, in: Proceedings of the 3rd International Conference on Genetic Algorithms, 1989, pp. 133–140.

[31] P. Poon, J. Carter, Genetic Algorithm Crossover Operators for Ordering Applications, Computers & Operations Research 22 (1) (1995) 135–147.

[32] J. Doe, Optimising PWR Reload Core Designs, in: R. Manner, B. Manderick (Eds.), Parallel Problem Solving from Nature 2, North Holland, 1992, p. 371.

[33] G. Parks, Multiobjective PWR Reload Core Optimization Using a Genetic Algorithm.

[34] A. Galperin, E. Nissan, Application of a Heuristic Search Method for Generation of Fuel Reload Configurations, Nucl. Sci. Eng. 99 (4).

[35] A. Galperin, S. Kimhi, M. Segev, A Knowledge-based System for Optimization of Fuel Reload Configurations, Nucl. Sci. Eng. 102 (1).

[36] E. Nissan, A. Galperin, Refueling in Nuclear Engineering: the FUELCON Project, Computers in Industry 37 (1) (1998) 43–54.

[37] J. Stevens, A Hybrid Method for In-Core Optimization of Pressurized Water Reactor Reload Core Design, Ph.D. thesis (1996).

[38] J. Stevens, K. Smith, K. Rempe, T. Downar, Optimization of Pressurized Water Reactor Shuffling by Simulated Annealing with Heuristics, Nuclear science and engineering 121 (1) (1995) 67–88.

[39] F. Verhagen, M. van der Schaar, W. de Kruijf, T. van de WETERING, R. Jones, ROSA: a utility tool for loading pattern optimization, in: Proc. of the ANS Topical Meeting–Advances in Nuclear Fuel Management II, Vol. 1, 1997, pp. 8–31.

[40] Studio360, foldit (Mar. 2009).
URL `http://www.studio360.org/2009/mar/20/foldit/`

[41] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popovic, et al., Predicting Protein Structures with a Multiplayer Online Game, Nature 466 (7307) (2010) 756–760.

pypgapack Documentation

pypgapack Documentation

Release 0.1.0

Jeremy Roberts

December 17, 2011

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH PYPGAPACK

1.1 Background

`pypgapack` is a Python wrapper for the parallel genetic algorithm library pgapack, written in C by David Levine. The source and documentation for pgapack can be found at http://ftp.mcs.anl.gov/pub/pgapack/. The motivation for wrapping the code is ultimately to support a class project aiming to optimize loading patterns of nuclear reactor cores, which is a rather large and difficult combinatorial problem. Lots of researchers have applied genetic algorithms (and many other algorithms) to the problem, and the class project aims to provide a flexible test bench in Python to investigate various ideas. Wrapping pgapack is one step toward that goal. pgapAck was chose largely due to limited but positive past experience with it.

It should be pointed out that a similar effort to wrap pgapack in Python was made called `pgapy` (see http://pgapy.sourceforge.net/), but I actually couldn't get it to work, probably because I didn't know a thing about building Python modules before I started this (and my minimal C knowledge didn't help matters). Hence, I decided to "roll my own" using SWIG in combination with a C++ wrapper around pgapack instead of interfacing directly with pgapack as `pgapy` does.

The `PGA` class wraps almost all of pgapack's functionality, including allowing user functions for several operations (like initialization, crossover, etc.) for the `PGA.DATATYPE_BINARY`, `PGA.DATATYPE_REAL` and `PGA.DATATYPE_INTEGER` alleles. No such support is currently offered for other allele types, including user-specified types. The intended way to use `pypgapack` is to derive classes from `PGA`, with objective and other functions as members.

Parallel functionality is supported with the help of mpi4py.

Note: `pypgapack` is currently in beta mode, so there may be many things that look wrapped but are not. Testing is a future goal, but not a priority—I need a grade! Feedback is welcome at robertsj@mit.edu.

1.2 Building pypgapack

Included in ./pypgapack are the required source files and a simple script `build_pypgapack` which generates the Python module. To build, do the following:

1. Build PGAPack with the patches in ./patches. The major difference is a slight change to allow use with C++. The Makefile template also is set to produce shared and static libraries.

2. Modify the paths and variables in `build_pypgapack` below to suit your needs.
3. The source as distributed is set for serial. To use in parallel, do the following:
    - Uncomment `PARALLEL` in `build_pypgapack`
    - Set `CXX` to the appropriate compiler (e.g. mpic++) in `build_pypgapack`
    - Delete or move the dummy `mpi.h` included with PGAPack to avoid redefinitions. There's probably a better approach.
    - This assumes PGAPack was built in parallel; if not, do so. Refer to the PGAPack documentation. You need an MPI-enabled compiler.
    - Get mpi4py (e.g. easy_install mpi4py). You need an MPI-enabled compiler. Note, a few files from mpi4py are included in ./pypgapack/mpi4py. These *may* need to be updated.
4. Execute `build_pypgapack` and set PYTHONPATH accordingly.

1.3 Next Steps

The user is encouraged to read the pgapack documentation thoroughly before using pypgapack, as the shared API is *not* covered in this documentation (and neither are the many PGAPack defaults). It's helpful to go through their examples in C/C++ or Fortran if you know the languages.

Thereafter, see the collection of *Examples*, which include several of the original pgapack examples along with a few additional ones that demonstrate how to use user-defined functions for a variety of operations. Reference output is included, though don't expect to reproduce the numbers exactly for the small number of generations used, as they'll be sensitive to compilation, etc.

For a quick refresher, the basic gist of genetic algorithms is discussed briefly in *Methods*, which lists a few references that may be of use.

Documentation for the relatively small number of additional methods not explicitly in pgapack can be found in the *API Reference*.

CHAPTER

TWO

EXAMPLES

All examples are located in the `pypgapack/examples` and the reference output for all examples is in `pypgapack/examples/output`. Aside from small floating point differences, the values should be the same given the use of a fixed random number generator seed in all the examples. A utility script `run_examples.py` is included to test user output to the included reference cases. (Note, the above might actually be untrue, as compilation can and will change how a fixed pseudo-random number sequence is generated.)

Also, the maximum generation count is limited to 50 for all cases to produce short output. Experiment with that limit to see better solutions.

2.1 Basic Examples

The following are some simple examples that illustrate the basic PGAPack functionality.

2.1.1 Example 1: MAXBIT

`pypgapack` is pretty easy to use, and to demonstrate, we'll solve the maxbit problem, the first example in the PGAPack documentation.

```
"""
pypgapack/examples/example01.py  --  maxbit
"""
from pypgapack import PGA
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxbit(self, p, pop) :
        """
        Maximum when all alleles are 1's, and that maximum is n.
        """
        val = 0
        # Size of the problem
        n = self.GetStringLength()
```

```
        for i in range(0, n) :
            # Check whether ith allele in string p is 1
            if self.GetBinaryAllele(p, pop, i) :
                val = val + 1
        # Remember that fitness evaluations must return a float
        return float(val)

# (Command line arguments, 1's and 0's, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_BINARY, 100, PGA.MAXIMIZE)
opt.SetRandomSeed(1)        # Set random seed for verification.
opt.SetMaxGAIterValue(50)   # 50 generations (default 1000) for short output.
opt.SetUp()                 # Internal allocations, etc.
opt.Run(opt.maxbit)         # Set the objective.
opt.Destroy()               # Clean up PGAPack internals
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       6.900000e+01
Iter #     Field      Value
20         Best       7.200000e+01
Iter #     Field      Value
30         Best       7.600000e+01
Iter #     Field      Value
40         Best       7.700000e+01
Iter #     Field      Value
50         Best       8.000000e+01
The Best Evaluation: 8.000000e+01.
The Best String:
[ 01111101111011110111111001110111101111011111110111111110100111111 ]
[ 01111111011111011111111110111101011011 ]
***Destroying PGA context***
```

2.1.2 Example 2: MAXINT

This is a similar problem, but the alleles are integers ranging from -100 to 100. Note that when the integer ranges are set, a cast to `intc` is used. Python uses high precision datatypes, and there doesn't seem to be a safe implicit conversion between the Python integer type and the C integer type behind the scenes (in SWIG land). Casting explicitly circumvents the issue.

```
"""
pypgapack/examples/example02.py  --  maxint
"""
from pypgapack import PGA
import numpy as np
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxint(self, p, pop) :
```

```
        """
        The maximum integer sum problem.

        The alleles are integers, and we solve
            max f(x) = x_1 + x_2 + ... + x_N
        subject to
            |x_i| <= 100 .
        That maximum is f(x) = 100n obtained for x_i = 100 for all i.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                       #   and sum it up.
        del c                                 # Delete "view" to internals.
        return float(val)                     # Always return a float.

n   = 10                # String length.
# (Command line arguments, integers, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)    # Set random seed for verification.
u_b =  100*np.ones(n)   # Define lower bound.
l_b = -100*np.ones(n)   # Define upper bound.
# Set the bounds.  Note, need to cast as C-combatible integers.
opt.SetIntegerInitRange(l_b.astype('intc'), u_b.astype('intc'))
opt.SetMaxGAIterValue(50)    # 50 generations for short output.
opt.SetUp()                  # Internal allocations, etc.
opt.Run(opt.maxint)          # Set the objective.
opt.Destroy()                # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       5.100000e+02
Iter #     Field      Value
20         Best       6.480000e+02
Iter #     Field      Value
30         Best       7.150000e+02
Iter #     Field      Value
40         Best       7.760000e+02
Iter #     Field      Value
50         Best       8.220000e+02
The Best Evaluation: 8.220000e+02.
The Best String:
#    0: [      27], [     100], [      86], [      98], [      97], [      99]
#    6: [      79], [      89], [      64], [      83]

***Destroying PGA context***
```

2.1.3 Example 3: MAXREAL

This is the same problem, but for real alleles. Here, note use of `SetMutationalType()` with the option of `PGA.MUTATION_RANGE`. This forces mutated allele values to remain within the initial range specified, useful for cases with constrained inputs. The default adds some (small) random amount, but over many

iterations, this can cause allele values to go significantly beyond the initial range.

```
"""
pypgapack/examples/example03.py  --  maxreal
"""
from pypgapack import PGA
import numpy as np
import sys

class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxreal(self, p, pop) :
        """
        The maximum real sum problem.

        The alleles are doubles, and we solve
        .. math::
          \max f(x) &= \sum^N_{n=1} x_n \\
              s.t.  &= |x_i| \leq 100
        That maximum is  :math:`f_{\textrm{max}}(x) = 100n` obtained for
        :math:`x_i = 100, i = 1\ldots N`.
        """
        c = self.GetRealChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                    #   and sum it up.
        del c                              # Delete "view" to internals.
        return val                         # Already a float.

n   = 10              # String length.
# (Command line arguments, doubles, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_REAL, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)  # Set random seed for verification.
u_b = 100*np.ones(n)  # Define lower bound.
l_b =-100*np.ones(n)  # Define upper bound.
# Set the bounds.  Default floats are handled without issue.
opt.SetRealInitRange(l_b, u_b)
# Force mutations to keep values in the initial range, a useful
#   feature for bound constraints.
opt.SetMutationType(PGA.MUTATION_RANGE)
opt.SetMaxGAIterValue(50) # 50 generations for short output.
opt.SetUp()               # Internal allocations, etc.
opt.Run(opt.maxreal)      # Set the objective.
opt.Destroy()             # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       5.535524e+02
Iter #     Field      Value
20         Best       6.550377e+02
Iter #     Field      Value
30         Best       7.378405e+02
```

```
Iter #     Field      Value
40         Best       7.537097e+02
Iter #     Field      Value
50         Best       7.827324e+02
The Best Evaluation: 7.827324e+02.
The Best String:
#   0: [   66.95374], [   47.92554], [    91.9023], [   75.87186], [   96.31829]
#   5: [   85.24209], [   69.58556], [   90.59017], [   86.26947], [   72.07333]

***Destroying PGA context***
```

2.2 Examples of User Defined Operators

These examples explore one of the strengths of PGAPack, namely user-defined operators.

2.2.1 Example 4: User-defined String Initialization

We redo *Example 2: MAXINT* by initializing the strings with our own routine. Here, that's done by generating a permutation using Numpy.

```
"""
pypgapack/examples/example04.py  --  maxint with user initialization
"""
from pypgapack import PGA
import numpy as np
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxint(self, p, pop) :
        """
        The maximum integer sum problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                       #   and sum it up.
        del c                                 # Delete "view" to internals.
        return float(val)                     # Always return a float.

    def init(self, p, pop) :
        """
        Random permutations using Numpy.
        """
        n    = self.GetStringLength()
        c    = self.GetIntegerChromosome(p, pop)
        perm = np.random.permutation(n)
        for i in range(0, n) :
            c[i] = perm[i]
        del c

```

```
n    = 10                # String length.
# (Command line arguments, integers, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)     # Set random seed for verification.
np.random.seed(1)        # Do the same with Numpy.
u_b =  100*np.ones(n)    # Define lower bound.
l_b = -100*np.ones(n)    # Define upper bound.
# Set the bounds.  Note, need to cast as C-compatible integers.
opt.SetIntegerInitRange(l_b.astype('intc'), u_b.astype('intc'))
opt.SetMaxGAIterValue(50)     # 50 generations for short output.
opt.SetUp()                   # Internal allocations, etc.
opt.Run(opt.maxint)           # Set the objective.
opt.Destroy()                 # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       5.100000e+02
Iter #     Field      Value
20         Best       6.480000e+02
Iter #     Field      Value
30         Best       7.150000e+02
Iter #     Field      Value
40         Best       7.760000e+02
Iter #     Field      Value
50         Best       8.220000e+02
The Best Evaluation: 8.220000e+02.
The Best String:
#    0: [      27], [     100], [      86], [      98], [      97], [      99]
#    6: [      79], [      89], [      64], [      83]

***Destroying PGA context***
```

2.2.2 Example 5: User-defined Crossover Operator

This example solves "Oliver's 30-city Hamiltonian cycle Traveling Salesman Problem", as described in Poon and Carter, *Computer Ops Res.*, **22**, (1995). More importantly, it demonstrates use of a user-defined crossover operator, namely the "Tie-Breaking Crossover" (TBX1) of the same work.

The problem has 30 cities in a plane, and the goal is to minimize the distance traveled when visiting each city just once in a complete loop. The problem has 40 equivalent optima, each with a distance of 423.741 units. The coordinates of the cities are in the code, and are from Oliver's original paper by way of Steve Dower's site. See *Parallel Examples* for a parallel version that tries matching the cited results.

```
"""
pypgapack/examples/example05.py  --  traveling salesman
"""
from pypgapack import PGA
import numpy as np
import sys

```

```
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def tsm(self, p, pop) :
        """
        Oliver's 30 city traveling salesman problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = self.distance(c)
        del c
        return val

    def distance(self, c) :
        """
        Compute the total distance for a set of cities.
        """
        # x and y coordinates by city
        x = np.array([54.0,54.0,37.0,41.0, 2.0, 7.0,25.0,22.0,18.0, 4.0,\
                      13.0,18.0,24.0,25.0,44.0,41.0,45.0,58.0,62.0,82.0,\
                      91.0,83.0,71.0,64.0,68.0,83.0,87.0,74.0,71.0,58.0])
        y = np.array([67.0,62.0,84.0,94.0,99.0,64.0,62.0,60.0,54.0,50.0,\
                      40.0,40.0,42.0,38.0,35.0,26.0,21.0,35.0,32.0, 7.0,\
                      38.0,46.0,44.0,60.0,58.0,69.0,76.0,78.0,71.0,69.0])
        n   = self.GetStringLength()
        val = 0.0
        for i in range(0, n-1) :
            val += np.sqrt( (x[c[i]]-x[c[i+1]])**2 + (y[c[i]]-y[c[i+1]])**2 )
        val += np.sqrt( (x[c[0]]-x[c[n-1]])**2 + (y[c[0]]-y[c[n-1]])**2 )
        assert(val > 423.70) # DEBUG.
        return val

    def tbx(self, p1, p2, pop1, c1, c2, pop2) :
        """
        Tie-breaking cross-over.  See Poon and Carter for details.
        """
        # Grab the city id's.
        paren1 = self.GetIntegerChromosome(p1, pop1)
        paren2 = self.GetIntegerChromosome(p2, pop1)
        child1 = self.GetIntegerChromosome(c1, pop2)
        child2 = self.GetIntegerChromosome(c2, pop2)
        assert(np.sum(paren1)==435) # DEBUG
        assert(np.sum(paren2)==435) # DEBUG

        # Copy the parents to temporary vector for manipulation.
        n = self.GetStringLength()
        parent1 = np.zeros(n)
        parent2 = np.zeros(n)
        for i in range(0, n) :
            parent1[i] = paren1[i]
            parent2[i] = paren2[i]

        # Code the parents using "position listing".
```

```
        code1   = np.zeros(n)
        code2   = np.zeros(n)
        for i in range(0, n) :
            code1[parent1[i]] = i + 1
            code2[parent2[i]] = i + 1

        # Randomly choose two cross-over points.
        perm   = np.random.permutation(n)
        point1 = np.min(perm[0:2])
        point2 = np.max(perm[0:2])+1

        # Exchange all alleles between the two points.  (It's unclear to me
        #   whether these points should be inclusive or not; here, they are.)
        temp = np.zeros(point2-point1)
        for i in range(point1, point2) :
            temp[i-point1] = parent1[i]
            parent1[i]     = parent2[i]
            parent2[i]     = temp[i-point1]

        # Generate a cross-over map, a random ordering of the 0,1,...,n-1
        crossovermap = np.random.permutation(n)

        # Multiply each allele of the strung by n and add the map.
        parent1 = parent1*n + crossovermap
        parent2 = parent2*n + crossovermap

        # Replace the lowest allele by 0, the next by 1, up to n-1.  Here,
        #   we sort the parents first, and then for each element, find
        #   where the increasing values are found in the original.  There
        #   is probably a simpler set of functions built in somewhere.
        sort1 = np.sort(parent1)
        sort2 = np.sort(parent2)
        for i in range(0, n) :
            index = np.where(parent1 == sort1[i])
            parent1[index[0][0]] = i
            index = np.where(parent2 == sort2[i])
            parent2[index[0][0]] = i

        # Map the string back to elements.  These are the offspring.
        tempchild1 = np.zeros(n)
        tempchild2 = np.zeros(n)
        for i in range(0, n) :
            tempchild1[parent1[i]] = i
            tempchild2[parent2[i]] = i
        for i in range(0, n) :
            child1[i] = tempchild1[i]
            child2[i] = tempchild2[i]

# Number of cities.
n    = 30

opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MINIMIZE)

```

```
114 # One possible benchmark solution
115 reference = np.array([ 0, 2, 3, 4, 5, 6, 7, 8, 9,10, \
116                       11,12,13,14,15,16,17,18,19,20, \
117                       21,22,24,23,25,26,27,28,29, 1] )
118 print "Reference distance is: ", opt.distance(reference)
119
120 opt.SetRandomSeed(1)                    # Set seed for verification.
121 np.random.seed(1)                       # Do the same with Numpy.
122 opt.SetIntegerInitPermute(0, n-1)       # Start with random permutations.
123 opt.SetPopSize(400)                     # Large enough to see some success.
124 opt.SetMaxGAIterValue(100)              # Small number for output.
125 opt.SetCrossover(opt.tbx)               # Set a cross-over operation.
126 opt.SetMutation(PGA.MUTATION_PERMUTE)   # Mutate by permutation.
127 opt.SetUp()                             # Internal allocations, etc.
128 opt.Run(opt.tsm)                        # Set the objective and run.
129 opt.Destroy()                           # Clean up PGAPack internals.
```

Running it once yields the following output:

```
***Constructing PGA***
Reference distance is:  423.740563133
Iter #     Field      Value
10         Best       1.012977e+03
Iter #     Field      Value
20         Best       9.924890e+02
Iter #     Field      Value
30         Best       9.924890e+02
Iter #     Field      Value
40         Best       9.924890e+02
Iter #     Field      Value
50         Best       9.419172e+02
Iter #     Field      Value
60         Best       8.709637e+02
Iter #     Field      Value
70         Best       8.709637e+02
Iter #     Field      Value
80         Best       8.524659e+02
Iter #     Field      Value
90         Best       8.524659e+02
Iter #     Field      Value
100        Best       7.805903e+02
The Best Evaluation: 7.805903e+02.
The Best String:
#    0: [      29], [       3], [       0], [      12], [      13], [       9]
#    6: [       4], [       8], [      11], [      10], [       5], [       6]
#   12: [       7], [      17], [      14], [      18], [      16], [      15]
#   18: [      26], [      24], [      22], [      19], [      21], [      25]
#   24: [      27], [      23], [      20], [      28], [       2], [       1]

***Destroying PGA context***
```

2.2.3 Example 6: User-defined Mutation Operator

We redo *Example 2: MAXINT* using a custom mutation operator, largely following the PGAPack example.

```
"""
pypgapack/examples/example06.py  --  maxint with user mutation.
"""
from pypgapack import PGA
import numpy as np
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxint(self, p, pop) :
        """
        The maximum integer sum problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = np.sum(c)                       #   and sum it up.
        del c                                 # Delete "view" to internals.
        return float(val)                     # Always return a float.

    def mutate(self, p, pop, pm) :
        """
        Mutate randomly within -n to n
        """
        n     = self.GetStringLength()
        c     = self.GetIntegerChromosome(p, pop)
        count = 0
        for i in range(0, n) :
            if self.RandomFlip(pm) :
                k = self.RandomInterval(1, 2*n)-n
                c[i] = k
                count += 1
        del c
        return count

n   = 10                # String length.
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)    # Set random seed for verification.
np.random.seed(1)       # Do the same with Numpy.
u_b =  100*np.ones(n)   # Define lower bound.
l_b = -100*np.ones(n)   # Define upper bound.
# Set the bounds.  Note, need to cast as C-combatible integers.
opt.SetIntegerInitRange(l_b.astype('intc'), u_b.astype('intc'))
opt.SetMaxGAIterValue(50)    # 50 generations for short output.
opt.SetMutation(opt.mutate)  # Set a custom mutation.
opt.SetUp()                  # Internal allocations, etc.
opt.Run(opt.maxint)          # Set the objective.
opt.Destroy()                # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       5.270000e+02
Iter #     Field      Value
20         Best       7.150000e+02
Iter #     Field      Value
30         Best       7.280000e+02
Iter #     Field      Value
40         Best       8.030000e+02
Iter #     Field      Value
50         Best       8.220000e+02
The Best Evaluation: 8.360000e+02.
The Best String:
#    0: [      67], [      65], [      94], [      89], [      85], [      99]
#    6: [      80], [      98], [      87], [      72]

***Destroying PGA context***
```

2.2.4 Example 7: User-defined End of Generation Operator

This example is almost the same as *Example 1: MAXBIT* but it adds an end-of-generation operator. Here, we're using it to flip a random bit to 1. Of course, since we're maximizing the bit sum, this is "climbing the hill" to a better answer. This is a trivial example of such heuristics; in other situations, there are more complex, physically-motivated approaches. Another use of an end-of-generation operator would be for post-generation processing, such as plotting fitnesses, writing to file, etc.

```
"""
pypgapack/examples/example07.py  --  maxbit with end-of-generation hill climb
"""
from pypgapack import PGA
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxbit(self, p, pop) :
        """
        Maximum when all alleles are 1's, and that maximum is n.
        """
        val = 0
        # Size of the problem
        n = self.GetStringLength()
        for i in range(0, n) :
            # Check whether ith allele in string p is 1
            if self.GetBinaryAllele(p, pop, i) :
                val = val + 1
        # Remember that fitness evaluations must return a float
        return float(val)
    def climb(self):
        """
        Randomly set a bit to 1 in each string
```

```
        """
        popsize = self.GetPopSize()
        n = self.GetStringLength()
        for p in range(0, popsize) :
            i = self.RandomInterval(0, n - 1)
            self.SetBinaryAllele(p, PGA.NEWPOP, i, 1)

# (Command line arguments, 1's and 0's, string length, and maximize it)
opt = MyPGA(sys.argv, PGA.DATATYPE_BINARY, 100, PGA.MAXIMIZE)
opt.SetRandomSeed(1)        # Set random seed for verification.
opt.SetMaxGAIterValue(50)   # 50 generations (default 1000) for short output.
opt.SetEndOfGen(opt.climb) # Set a hill climbing heuristic
opt.SetUp()                 # Internal allocations, etc.
opt.Run(opt.maxbit)         # Set the objective.
opt.Destroy()               # Clean up PGAPack internals
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       7.000000e+01
Iter #     Field      Value
20         Best       7.600000e+01
Iter #     Field      Value
30         Best       8.300000e+01
Iter #     Field      Value
40         Best       8.800000e+01
Iter #     Field      Value
50         Best       9.100000e+01
The Best Evaluation: 9.300000e+01.
The Best String:
[ 11111111111111111110111111111111111111011111011011111011111110111 ]
[ 11111111111111111111111101111111111 ]
***Destroying PGA context***
```

Notice that for the same settings, the heuristic improved the best solution a little bit. Of course, flipping just one bit out of 100 shouldn't be expected to work miracles.

2.3 Parallel Examples

The following examples illustrate use of `pypgapack` in a parallel setting using the `mpi4py` package.

2.3.1 Example 8: Parallel MAXBIT

We adapt our favorite *Example 1: MAXBIT* using MPI. We up the string length and population to bring out timing differences.

```
"""
pypgapack/examples/example08.py  --  parallel maxbit
"""
```

```
from pypgapack import PGA
from mpi4py import MPI
import sys
class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def maxbit(self, p, pop) :
        """
        Maximum when all alleles are 1's, and that maximum is n.
        """
        val = 0
        # Size of the problem
        n = self.GetStringLength()
        for i in range(0, n) :
            # Check whether ith allele in string p is 1
            if self.GetBinaryAllele(p, pop, i) :
                val = val + 1
        # Remember that fitness evaluations must return a float
        return float(val)

comm = MPI.COMM_WORLD           # Get the communicator.
rank = comm.Get_rank()          # Get my rank.
if rank == 0 :                  # Just to show it works, have node 0
    seed = 1                    #   set seed=1 and n=500 and have all
    n    = 500
else :                          #   other nodes set seed=n=0.  Then,
    seed = 0                    #   broadcast them to all nodes with
    n    = 0
seed = comm.bcast(seed, root=0) #   node 0 as the root process,
n    = comm.bcast(n,    root=0) #   and verify by printing.
print " node=", rank, " seed=", seed, " n=", n

if rank == 0 :
    t_start = MPI.Wtime()  # Start the clock.
# (Command line arguments, 1's and 0's, string length, and maximize it)
opt  = MyPGA(sys.argv, PGA.DATATYPE_BINARY, n, PGA.MAXIMIZE)
opt.SetPopSize(1000)
opt.SetRandomSeed(seed)         # Set random seed for verification.
opt.SetMaxGAIterValue(50)       # 50 generations for short output.
opt.SetUp()                     # Internal allocations, etc.
opt.Run(opt.maxbit)             # Set the objective.
opt.Destroy()                   # Clean up PGAPack internals
if rank == 0 :
    t_end = MPI.Wtime()
    print "Elapsed time = ", t_end-t_start, " seconds."
MPI.Finalize() # Should be called automatically, but good practice.
```

Running it using `mpirun -np 1 python example08.py` yields the following output:

```
 node= 0  seed= 1  n= 500
***Constructing PGA***
Iter #     Field      Value
```

```
10         Best       2.930000e+02
Iter #     Field      Value
20         Best       3.030000e+02
Iter #     Field      Value
30         Best       3.080000e+02
Iter #     Field      Value
40         Best       3.240000e+02
Iter #     Field      Value
50         Best       3.280000e+02
The Best Evaluation: 3.290000e+02.
The Best String:
[ 1001010110110110111110111111011001010111010111101100011011100011 ]
[ 1111111101101111111110100101100011101110011111110110110011110001 ]
[ 1010100111101111111111101110101100111111110101100110011111111001 ]
[ 0101011111111100011111111111011011101100111101111011101110010110000 ]
[ 1011110100011110010110010110011011111001010001001011000101101111 ]
[ 0111101101111100111101011100111111010110111111100101110010110111 ]
[ 1011000101110001111100011111110111110100101110000010010011100010 ]
[ 1011101100101110111001110000011111111111101101111111 ]
***Destroying PGA context***
Elapsed time =  2.06364393234  seconds.
```

Running it using `mpirun -np 2 python example08.py` yields the following output:

```
 node= 0  seed= 1  n= 500
***Constructing PGA***
 node= 1  seed= 1  n= 500
***Constructing PGA***
Iter #     Field      Value
10         Best       2.930000e+02
Iter #     Field      Value
20         Best       3.030000e+02
Iter #     Field      Value
30         Best       3.080000e+02
Iter #     Field      Value
40         Best       3.240000e+02
Iter #     Field      Value
50         Best       3.280000e+02
***Destroying PGA context***
The Best Evaluation: 3.290000e+02.
The Best String:
[ 1001010110110110111110111111011001010111010111101100011011100011 ]
[ 1111111101101111111110100101100011101110011111110110110011110001 ]
[ 1010100111101111111111101110101100111111110101100110011111111001 ]
[ 0101011111111100011111111111011011101100111101111011101110010110000 ]
[ 1011110100011110010110010110011011111001010001001011000101101111 ]
[ 0111101101111100111101011100111111010110111111100101110010110111 ]
[ 1011000101110001111100011111110111110100101110000010010011100010 ]
[ 1011101100101110111001110000011111111111101101111111 ]
***Destroying PGA context***
Elapsed time =  1.14050102234  seconds.
```

This was using a dual core laptop with probably more browser windows open than needed, but the results

look good. Overall, PGAPack focuses parallelism on the object function evaluation. Hence, if your objective function is expensive to evaluate, you can expect relatively good scaling up to the number of strings replaced every generation (the default is 1/10 of the total).

2.3.2 Example 9: Parallel Traveling Salesman

We adapt the Traveling Salesman Problem to parallel and try matching the results Poon and Carter found for the TBX1 cross-over operator. The GA parameters used by Poon and Carter are not entirely clear. They cite results for a population of 21 over 300 iterations with a cross-over to mutation probability ratio of 0.8/0.2. The exact nature of the "swap" mutation is cited from another work I don't have at hand, and the selection appears to be standard elitist, i.e all but the best is replaced. I set a swap operator that always swaps a user-set number of pairs. Using 3 pairs (i.e. 10%) seems to get close to the cited results. Also, because PGAPack doesn't like odd population sizes, I use 22.

```
"""
pypgapack/examples/example09.py  --  traveling salesman in parallel
"""
from pypgapack import PGA
from mpi4py import MPI
import numpy as np
import sys

class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def tsm(self, p, pop) :
        """
        Oliver's 30 city traveling salesman problem.
        """
        c = self.GetIntegerChromosome(p, pop) # Get pth string as Numpy array
        val = self.distance(c)
        del c
        return val

    def distance(self, c) :
        """
        Compute the total distance for a set of cities.
        """
        # x and y coordinates by city
        x = np.array([54.0,54.0,37.0,41.0, 2.0, 7.0,25.0,22.0,18.0, 4.0,\
                      13.0,18.0,24.0,25.0,44.0,41.0,45.0,58.0,62.0,82.0,\
                      91.0,83.0,71.0,64.0,68.0,83.0,87.0,74.0,71.0,58.0])
        y = np.array([67.0,62.0,84.0,94.0,99.0,64.0,62.0,60.0,54.0,50.0,\
                      40.0,40.0,42.0,38.0,35.0,26.0,21.0,35.0,32.0, 7.0,\
                      38.0,46.0,44.0,60.0,58.0,69.0,76.0,78.0,71.0,69.0])
        n   = self.GetStringLength()
        val = 0.0
        for i in range(0, n-1) :
            val += np.sqrt( (x[c[i]]-x[c[i+1]])**2 + (y[c[i]]-y[c[i+1]])**2 )
        val += np.sqrt( (x[c[0]]-x[c[n-1]])**2 + (y[c[0]]-y[c[n-1]])**2 )
        assert(val > 423.70) # Debug.
```

```
        return val

    def tbx(self, p1, p2, pop1, c1, c2, pop2) :
        """
        Tie-breaking cross-over.  See Poon and Carter for details.
        """
        # Grab the city id's.
        paren1 = self.GetIntegerChromosome(p1, pop1)
        paren2 = self.GetIntegerChromosome(p2, pop1)
        child1 = self.GetIntegerChromosome(c1, pop2)
        child2 = self.GetIntegerChromosome(c2, pop2)
        assert(np.sum(paren1)==435) # DEBUG
        assert(np.sum(paren2)==435) # DEBUG

        # String length.
        n = self.GetStringLength()
        parent1 = np.zeros(n)
        parent2 = np.zeros(n)

        for i in range(0, n) :
            parent1[i] = paren1[i]
            parent2[i] = paren2[i]

        # Code the parents using "position listing".
        code1   = np.zeros(n)
        code2   = np.zeros(n)
        for i in range(0, n) :
            code1[parent1[i]] = i + 1
            code2[parent2[i]] = i + 1

        # Randomly choose two cross-over points.
        perm   = np.random.permutation(n)
        point1 = np.min(perm[0:2])
        point2 = np.max(perm[0:2])+1

        # Exchange all alleles between the two points.
        temp = np.zeros(point2-point1)
        for i in range(point1, point2) :
            temp[i-point1] = parent1[i]
            parent1[i]     = parent2[i]
            parent2[i]     = temp[i-point1]

        # Generate a cross-over map, a random ordering of the 0,1,...,n-1
        crossovermap = np.random.permutation(n)

        # Multiply each allele of the strung by n and add the map.
        parent1 = parent1*n + crossovermap
        parent2 = parent2*n + crossovermap

        # Replace the lowest allele by 0, the next by 1, up to n-1.
        sort1 = np.sort(parent1)
        sort2 = np.sort(parent2)
        for i in range(0, n) :
```

```
92              index = np.where(parent1 == sort1[i])
93              parent1[index[0][0]] = i
94              index = np.where(parent2 == sort2[i])
95              parent2[index[0][0]] = i
96
97          tmpc1 = np.zeros(n)
98          tmpc2 = np.zeros(n)
99          # Map the string back to elements.  These are the offspring.
100         for i in range(0, n) :
101             tmpc1[parent1[i]] = i
102             tmpc2[parent2[i]] = i
103         for i in range(0, n) :
104             child1[i] = tmpc1[i]
105             child2[i] = tmpc2[i]
106
107     def swap(self, p, pop, pm) :
108         """
109         Random swap of allele pairs.  Note, nswap must be set!
110         """
111         n     = self.GetStringLength()
112         c     = self.GetIntegerChromosome(p, pop)
113         index = np.random.permutation(n)
114         for i in range(0, self.nswap) :
115             i1 = index[2*i  ]
116             i2 = index[2*i+1]
117             tmp1      = c[i1]
118             tmp2      = c[i2]
119             c[i1]     = tmp2
120             c[i2]     = tmp1
121         del c
122         return 0
123
124     def init(self, p, pop) :
125         """
126         Random initial states.  We do this so that we can enforce the same
127         initial guesses for all runs to compare against the Poon and Carter.
128         """
129         n = self.GetStringLength()
130         c = self.GetIntegerChromosome(p, pop)
131         np.random.seed(p)
132         perm = np.random.permutation(n)
133         for i in range(0, n) :
134             c[i] = perm[i]
135         del c
136
137 comm = MPI.COMM_WORLD           # Get the communicator.
138 rank = comm.Get_rank()          # Get my rank.
139 t_start = MPI.Wtime()           # Start the clock.
140 n        = 30                   # Number of cities.
141 numrun   = 25                   # Number of runs to average.
142 besteval = np.zeros(numrun)
143 for i in range(0, numrun) :
144     opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MINIMIZE)
```

```
145     opt.SetInitString(opt.init) # Set an initialization operator.
146     opt.SetCrossoverProb(0.8)   # (Default is 85%)
147     opt.SetPopSize(22)          # 22 rather than 21
148     opt.SetNumReplaceValue(21)  # Keep the best 22-21 = 1 string = elitist.
149     opt.SetMaxGAIterValue(300)  # 300 generations, like the reference.
150     opt.SetCrossover(opt.tbx)   # Set a cross-over operation.
151     opt.SetMutation(opt.swap)   # Set a mutate operation.
152     opt.nswap = 3               # Number of pairs to swap in mutation.
153     opt.SetUp()                 # Internal allocations, etc.
154     opt.Run(opt.tsm)            # Set the objective and run.
155     if rank == 0 :
156         best        = opt.GetBestIndex(PGA.OLDPOP)
157         besteval[i] = opt.GetEvaluation(best, PGA.OLDPOP)
158     opt.Destroy()              # Clean up PGAPack internals.
159
160 if rank == 0 :
161     print "  MEAN: ", np.mean(besteval) # Print out the mean
162     print " SIGMA: ", np.std(besteval)  # and standard deviation.
163     print "   MIN: ", np.min(besteval)  # Print out the mean
164     print "   MAX: ", np.max(besteval)  # Print out the mean
165     t_end = MPI.Wtime()
166     print "Elapsed time = ", t_end-t_start, " seconds."
167
168 MPI.Finalize() # Should be called automatically, but good practice.
```

Running it using `mpirun -np 1 python example09.py` yields the following output (last several lines):

```
300        Best       9.143837e+02
The Best Evaluation: 9.143837e+02.
The Best String:
#    0: [       7], [      29], [       5], [       2], [      28], [      22]
#    6: [       8], [      10], [      18], [      19], [       3], [      15]
#   12: [      14], [      27], [      16], [       4], [      24], [      20]
#   18: [      17], [      12], [      26], [      25], [       0], [       9]
#   24: [       6], [      13], [      23], [      11], [      21], [       1]

***Destroying PGA context***
  MEAN:  844.962788368
 SIGMA:  97.2494217692
   MIN:  622.625633555
   MAX:  997.810825333
Elapsed time =  67.2148938179  seconds.
```

Running it using `mpirun -np 6 python example09.py` yields the following output (last several lines):

```
Iter #     Field      Value
300        Best       8.004605e+02
The Best Evaluation: 8.004605e+02.
The Best String:
#    0: [       7], [      18], [      17], [      14], [      25], [       5]
#    6: [      19], [      24], [      16], [       2], [       3], [      22]
```

```
#   12: [      21], [       1], [      23], [      26], [       8], [      28]
#   18: [       9], [      12], [      11], [      15], [      20], [      10]
#   24: [      27], [      29], [      13], [       0], [       4], [       6]

***Destroying PGA context***
  MEAN:  832.288188184
 SIGMA:  79.9132513415
   MIN:  597.338150861
   MAX:  942.115139223
Elapsed time =  45.1982860565  seconds.
```

This was using a 6-core machine, but the speedup was barely 25%—why? Well, evaluating the distance of 30 cities is cheap compared to the sorting occuring on the master process for the cross-over operation. As noted earlier, PGAPack's parallelism is definitely meant for expensive evaluations relative to everything else. Here, we see *some* speedup, just not very good. Also compare the mean and standard deviation to the cited 829.00 and 72.69. The parallel results are much closer, which may be a fluke, but it may be worth investigating.

2.3.3 Example 10: Optimizing a Slab Reactor

In this problem, the goal is to optimize a "slab reactor". While the physics is out of our scope, the essential idea is as follows. We have 8 slabs, each 20 cm thick, and each of a different "fuel". These slabs are situated together in some pattern. The pattern is surrounded on either side by water, and beyond the right side water is vacuum while on the left side is an effective mirror, i.e. what goes in must return. This definition makes it so the sequence [0,1,...,7] is different from [7,...,1,0]. What we want to do is to maximize the multiplication factor "keff" (which is related to how long the reactor could produce energy before needing more fuel) and make the power distribution as flat as possible (which in real reactors is related to several important safety margins). The latter is quantified by the "peaking factor", defined as the maximum assembly power divided by the average power of all assemblies, and we seek to minimize this. The objective function is a weighted sum of these objectives.

The crossover used is the heuristic tie-breaking crossover (HTBX) described in Carter, *Advances in Nuclear Science and Technology*., **25**, (1997). The basic idea is similar to the TBX operator of the TSP examples but includes extra problem information, in this case the "reactivity" of the fuel as quantified by "k-infinity". Basically, a more "reactive" fuel produces more neutrons with time, or, in other words, contributes more to the energy production of the reactor. Because of the way the materials are ordered in this example, they are already sorted by reactivity. Hence, TBX and HTBX are identical in implementation for this case.

We run the problem over 100 generations and with a population of size 50. We employ a rather elitist strategy, replacing 40 strings each generation. We also disallow identical strings.

```
"""
pypgapack/examples/example10.py  --  optimize a reflected 8 slab reactor
"""

from pypgapack import PGA
from mpi4py import MPI
import numpy as np
import sys
import matplotlib.pyplot as plt
```

```
from scipy import factorial
from matplotlib import rc
rc('text', usetex=True)
rc('font', family='serif')

class MyPGA(PGA) :
    """
    Derive our own class from PGA.
    """
    def f(self, p, pop) :
        """
        Minimize peaking and maximize keff using weighted objective.
        """
        pattern = self.GetIntegerChromosome(p, pop)
        keff, peak = self.flux(pattern)
        del pattern
        delta = 0
        if peak > 1.8 :
            delta = peak - 1.8
        self.evals += 1
        return 1.0*keff - 10.0*delta

    def flux(self, pattern, plot=0) :
        """
        Solve for the flux via simple mesh-centered finite differences.

        Returns keff and the maximum-to-average assembly-averaged
        fission density ratio.
        """
        # Coarse mesh boundaries.  (Specific to n=8!!)
        coarse = np.array([0.,20.,40.,60.,80.,100.,120.,140.,160.,180.,200.])
        # Fine meshes per coarse mesh.
        fine = 20
        dx = 20.0 / fine
        # Material map (simplifies coefficients)
        mat = np.zeros(fine * (len(coarse) - 1))
        mat[0:fine] = 10 # reflector
        j = fine
        for i in range(0, self.number_slabs) :
            mat[j:(j + fine)] = pattern[i] # one of the slabs
            j += fine
        mat[j:(j + fine)] = 10 # reflector
        # System size
        n = fine * (len(coarse) - 1)
        # Materials
        D, R, F, S = self.materials()
        # Coefficient Matrix (just diagonals) and Vectors
        AD = np.zeros((n, 2))
        AL = np.zeros((n, 2))
        AU = np.zeros((n, 2))
        nufission = np.zeros((n, 2))
        scatter12 = np.zeros(n)
        for g in range(0, 2) :
```

```
            # reflective left boundary
            b = 1
            AU[0, g] = -2.0 * D[g, mat[0]] * D[g, mat[0]] / \
                       (D[g, mat[0]] * dx + D[g, mat[1]] * dx)
            AD[0, g] = -AU[0, g] + 2.0 * D[g, mat[0]] * (1.0 - b) / \
                       (4.0 * D[g, mat[0]] * (1.0 + b) + dx * (1.0 - b)) + \
                       dx * R[g, mat[0]]
            nufission[0, g] = dx * F[g, mat[0]]
            scatter12[0] = dx * S[mat[0]]
            # vacuum right boundary
            b = 0
            AL[n - 2, g] = -2.0 * D[g, mat[n - 1]] * D[g, mat[n - 2]] / \
                           (D[g, mat[n - 1]] * dx + D[g, mat[n - 2]] * dx)
            AD[n - 1, g] = -AL[n - 2, g] + \
                        2.0 * D[g, mat[n - 1]] * (1.0 - b) / \
                       (4.0 * D[g, mat[n - 1]] * (1.0 + b) + \
                       dx * (1.0 - b)) + \
                       dx * R[g, mat[n - 1]]
            nufission[n - 1, g] = dx * F[g, mat[n - 1]]
            scatter12[n - 1] = dx * S[mat[n - 1]]
            # internal cells
            for i in range(1, n - 1) :
                AL[i - 1, g] = -2.0 * D[g, mat[i]] * D[g, mat[i - 1 ]] / \
                               (D[g, mat[i]] * dx + D[g, mat[i - 1]] * dx)
                AU[i, g] = -2.0 * D[g, mat[i]] * D[g, mat[i + 1 ]] / \
                           (D[g, mat[i]] * dx + D[g, mat[i + 1]] * dx)

                AD[i, g] = -(AL[i - 1, g] + AU[i, g]) + dx * R[g, mat[i]]
                nufission[i, g] = dx * F[g, mat[i]]
                scatter12[i] = dx * S[mat[i]]
        # Initiate the fluxes.
        phi1 = np.zeros(n)
        phi2 = np.zeros(n)
        # Use a guess for fission density based on fission cross section.
        fission_density = nufission[:, 1]
        # and normalize it.
        fission_density = fission_density / np.sqrt(np.sum(fission_density**2))
        fission_density0 = np.zeros(n)
        # Initialize the downscatter source.
        scatter_source = np.zeros(n)
        # Initial eigenvalue guess.
        keff = 1
        keff0 = 0
        # Set errors.
        errorfd = 1.0
        errork = 1.0
        it = 0
        while (errorfd > 1e-5 and errork > 1e-5 and it < 200) :
            # Solve fast group.
            self.tridiag(AU[:, 0], AL[:, 0], AD[:, 0], \
                         fission_density / keff, phi1)
            # Compute down scatter source.
            scatter_source = phi1 * scatter12
```

```
116             # Solve thermal group.
117             self.tridiag(AU[:, 1], AL[:, 1], AD[:, 1], scatter_source, phi2)
118             # Keep old values.
119             fission_density0[:] = fission_density
120             keff0 = keff
121             # Update density and eigenvalue
122             fission_density[:] = phi1 * nufission[:, 0] + \
123                                  phi2 * nufission[:, 1]
124             keff = keff0 * np.sum(fission_density) / \
125                            np.sum(fission_density0)
126             # Update errors.  Use Linf norm on density.
127             errorfd = np.max(np.abs(fission_density - fission_density0))
128             errork = np.abs(keff - keff0)
129             it += 1
130         # Now we average the fission density over each fueled coarse mesh.
131         slab_fission_density = np.zeros(self.number_slabs)
132         j = fine
133         for i in range(0, self.number_slabs) :
134             slab_fission_density[i] = \
135                 np.mean(fission_density[j:(j + fine) - 1])
136             j += fine
137         mean_fission_density = np.mean(fission_density)
138         peaking = slab_fission_density / mean_fission_density
139         max_peaking = np.max(peaking)
140         return keff, max_peaking
141 
142     def tridiag(self, U, L, D, f, y):
143         """
144         Tridiagonal solver.
145 
146         This assumes vectors U, L, and D are of the same length.  The right
147         hand side is f and the solved unknowns are returned in y.
148         """
149         N = len(D)
150         w = np.zeros(N)
151         v = np.zeros(N)
152         z = np.zeros(N)
153         w[0] = D[0]
154         v[0] = U[0] / w[0]
155         z[0] = f[0] / w[0]
156         for i in range(1, N) :
157             w[i] = D[i] - L[i - 1] * v[i - 1]
158             v[i] = U[i] / w[i]
159             z[i] = (f[i] - L[i - 1] * z[i - 1]) / w[i]
160         y[N - 1] = z[N - 1]
161         for i in range(N - 2, -1, -1) :
162             y[i] = z[i] - v[i] * y[i + 1]
163 
164     def materials(self):
165         """
166         10 fuels with one 1 reflector by row.  Represents burnup of 0, 5,
167         10, 15, 20, 25, 30, 35, 40, and 45 MWd/kg for one assembly type.
168         """
```

```
169         D = np.array([                                                  \
170             [1.4402e+00, 1.4429e+00, 1.4453e+00, 1.4467e+00, 1.4476e+00,\
171              1.4483e+00, 1.4489e+00, 1.4496e+00, 1.4507e+00, 1.4525e+00,\
172              1.3200e+00],                                               \
173             [3.7939e-01, 3.7516e-01, 3.7233e-01, 3.7045e-01, 3.6913e-01,\
174              3.6818e-01, 3.6749e-01, 3.6699e-01, 3.6649e-01, 3.6615e-01,\
175              2.6720e-01]])
176         R = np.array([                                                  \
177             [2.5800e-02, 2.5751e-02, 2.5755e-02, 2.5840e-02, 2.5958e-02,\
178              2.6090e-02, 2.6226e-02, 2.6362e-02, 2.6559e-02, 2.6802e-02,\
179              2.5700e-02],                                               \
180             [1.1817e-01, 1.2301e-01, 1.2306e-01, 1.2277e-01, 1.2223e-01,\
181              1.2136e-01, 1.2017e-01, 1.1871e-01, 1.1622e-01, 1.1275e-01,\
182              5.1500e-02]])
183         F = np.array([                                                  \
184             [7.9653e-03, 7.6255e-03, 7.2724e-03, 6.9344e-03, 6.6169e-03,\
185              6.3189e-03, 6.0453e-03, 5.7908e-03, 5.4413e-03, 5.0379e-03,\
186              0.00000000],                                               \
187             [1.6359e-01, 1.7301e-01, 1.7681e-01, 1.7634e-01, 1.7355e-01,\
188              1.6931e-01, 1.6424e-01, 1.5869e-01, 1.5005e-01, 1.3894e-01,\
189              0.00000000]])
190         S = np.array(                                                   \
191             [1.5204e-02, 1.5152e-02, 1.4958e-02, 1.4828e-02, 1.4744e-02,\
192              1.4691e-02, 1.4652e-02, 1.4626e-02, 1.4601e-02, 1.4587e-02,\
193               2.3100e-02])
194         return D, R, F, S
195
196     def kinf(self, pattern) :
197         """
198         Compute kinf for each slab.
199         """
200         D, R, F, S = self.materials()
201         k = np.zeros(len(pattern))
202         for i in range(0, len(pattern)) :
203             k[i] = (F[0, i] + F[1, i] * S[i] / R[1, i]) / R[0, i]
204         return k
205
206     def htbx(self, p1, p2, pop1, c1, c2, pop2) :
207         """
208         Heuristic tie-breaking cross-over.  See Carter for details. Actually,
209         for this problem, HTBX is equivalent to TBX, since the materials are
210         already ordered by reactivity.
211         """
212         # Grab the city id's.
213         paren1 = self.GetIntegerChromosome(p1, pop1)
214         paren2 = self.GetIntegerChromosome(p2, pop1)
215         child1 = self.GetIntegerChromosome(c1, pop2)
216         child2 = self.GetIntegerChromosome(c2, pop2)
217         # Copy the parents to temporary vector for manipulation.
218         n = self.GetStringLength()
219         parent1 = np.zeros(n)
220         parent2 = np.zeros(n)
221         for i in range(0, n) :
```

```
            parent1[i] = paren1[i]
            parent2[i] = paren2[i]
        # Code the parents using "position listing".
        code1 = np.zeros(n)
        code2 = np.zeros(n)
        for i in range(0, n) :
            code1[parent1[i]] = i + 1
            code2[parent2[i]] = i + 1
        # Randomly choose two cross-over points.
        perm = np.random.permutation(n)
        point1 = np.min(perm[0:2])
        point2 = np.max(perm[0:2]) + 1
        # Exchange all alleles between the two points.
        temp = np.zeros(point2 - point1)
        for i in range(point1, point2) :
            temp[i - point1] = parent1[i]
            parent1[i] = parent2[i]
            parent2[i] = temp[i - point1]
        # Generate a cross-over map, a random ordering of the 0,1,...,n-1
        crossovermap = np.random.permutation(n)
        # Multiply each allele of the strung by n and add the map.
        parent1 = parent1 * n + crossovermap
        parent2 = parent2 * n + crossovermap
        # Replace the lowest allele by 0, the next by 1, up to n-1.  Here,
        #   we sort the parents first, and then for each element, find
        #   where the increasing values are found in the original.  There
        #   is probably a simpler set of functions built in somewhere.
        sort1 = np.sort(parent1)
        sort2 = np.sort(parent2)
        for i in range(0, n) :
            index = np.where(parent1 == sort1[i])
            parent1[index[0][0]] = i
            index = np.where(parent2 == sort2[i])
            parent2[index[0][0]] = i
        # Map the string back to elements.  These are the offspring.
        tempchild1 = np.zeros(n)
        tempchild2 = np.zeros(n)
        for i in range(0, n) :
            tempchild1[parent1[i]] = i
            tempchild2[parent2[i]] = i
        for i in range(0, n) :
            child1[i] = tempchild1[i]
            child2[i] = tempchild2[i]

    def eog(self):
        """
        Log some data for each generation.
        """
        best                = self.GetBestIndex(PGA.OLDPOP)
        bestpattern         = self.GetIntegerChromosome(best, PGA.OLDPOP)
        iter                = self.GetGAIterValue()
        keff, peak          = opt.flux(bestpattern)
        self.besteval[iter-1] = self.GetEvaluation(best, PGA.OLDPOP)
```

```
        self.bestkeff[iter-1] = keff
        self.bestpeak[iter-1] = peak

comm = MPI.COMM_WORLD           # Get the communicator.
rank = comm.Get_rank()          # Get my rank.
size = comm.Get_size()
t_start = MPI.Wtime()           # Start the clock.
n = 8                           # Number of fueled slabs (1-10)
opt = MyPGA(sys.argv, PGA.DATATYPE_INTEGER, n, PGA.MAXIMIZE)
opt.SetRandomSeed(1)                    # Set seed for verification.
np.random.seed(1)                       # Do the same with Numpy.
opt.SetIntegerInitPermute(0, n - 1)     # Start with random permutations.
opt.SetPopSize(50)                      # Large enough to see some success.
opt.SetNumReplaceValue(40)              # Keep the best half.
opt.SetMaxGAIterValue(50)               # Small number for output.
opt.SetCrossover(opt.htbx)              # Set a cross-over operation.
opt.SetEndOfGen(opt.eog)                # End of generation info
opt.SetMutation(PGA.MUTATION_PERMUTE)   # Mutate by permutation.
opt.SetNoDuplicatesFlag(PGA.TRUE)       # Keep no duplicate patterns.
opt.SetUp()                             # Internal allocations, etc.
opt.number_slabs = n                    # Must be set to string size.
opt.evals = 0
opt.besteval = np.zeros(51)
opt.bestkeff = np.zeros(51)
opt.bestpeak = np.zeros(51)
opt.Run(opt.f)

if rank > 0:
    evals = np.array([opt.evals],dtype='i')
    comm.Send([evals,MPI.INT], dest=0, tag=13)
else :
    evals = np.array([1],dtype='i')
    for i in range(1, size) :
        comm.Recv([evals,MPI.INT], source=i, tag=13)
        opt.evals += evals[0]

if rank == 0 :
    best = opt.GetBestIndex(PGA.OLDPOP) # Get the best string
    bestpattern = opt.GetIntegerChromosome(best, PGA.OLDPOP)
    keff, peak = opt.flux(bestpattern)  #   and its keff and peak
    print " best keff = ", keff, " and peak = ", peak
    t_end = MPI.Wtime()
    print "Elapsed time = ", t_end-t_start, " seconds."
    print "# Evaluations = ", opt.evals
    plt.plot( np.arange(0,51), opt.besteval, 'b',                          \
              np.array([0,51]), np.array([1.06611227 , 1.06611227]), 'g--',\
              lw=2) # Plot the objective as a function of generations
                    #   against the reference solution.
    plt.title('Convergence of Objective')
    plt.xlabel(' generation')
    plt.ylabel(' objective ')
    plt.legend(('HTBX', 'solution'),loc=4, shadow=True)
    plt.grid(True)
```

```
328     plt.show()
329
330 MPI.Finalize()
331 opt.Destroy()
```

Running it using `mpirun -np 1 python example10.py` yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       1.060239e+00
Iter #     Field      Value
20         Best       1.065283e+00
Iter #     Field      Value
30         Best       1.065808e+00
Iter #     Field      Value
40         Best       1.065808e+00
Iter #     Field      Value
50         Best       1.065808e+00
The Best Evaluation: 1.065808e+00.
The Best String:
#    0: [       4], [       2], [       1], [       5], [       3], [       0]
#    6: [       7], [       6]

 best keff =  1.06416554234  and peak =  2.00346511469
Elapsed time =  179.573677063  seconds.
# Evaluations =  1990
***Destroying PGA context***
```

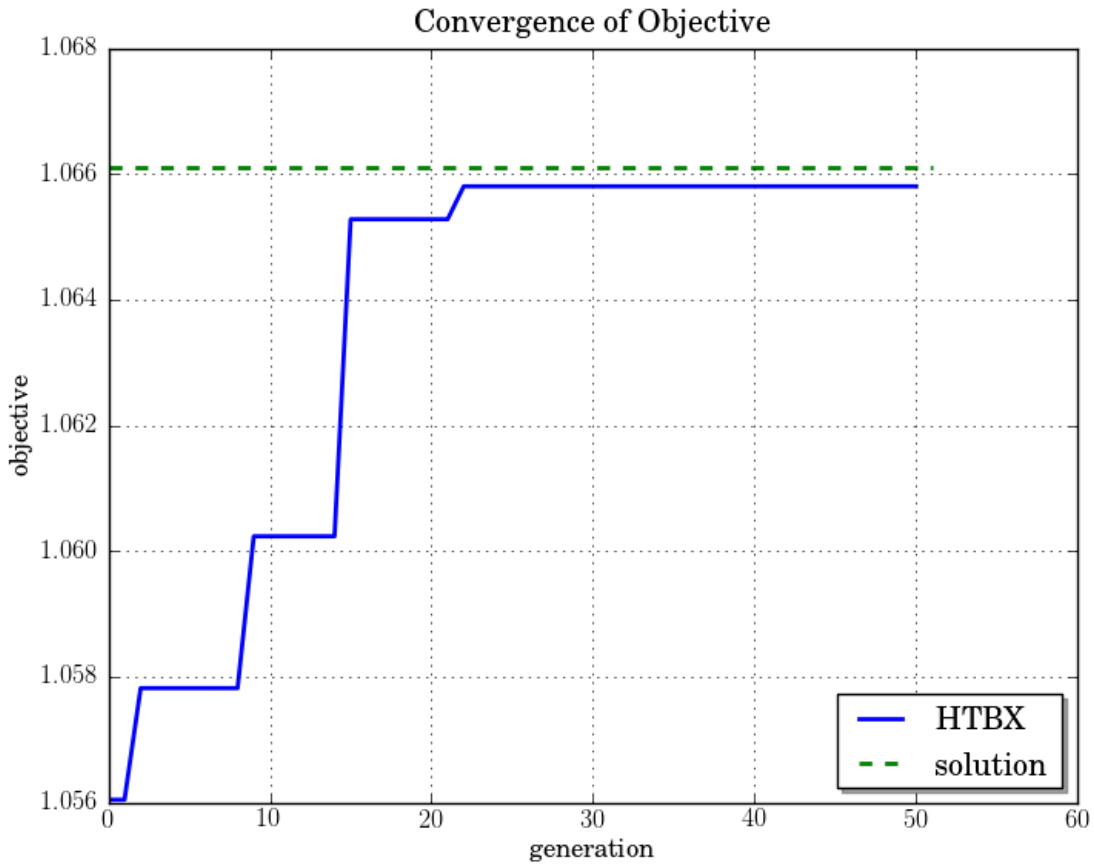
Running it using `mpirun -np 2 python example10.py` yields the following output:

```
***Constructing PGA***
***Constructing PGA***
Iter #     Field      Value
10         Best       1.060239e+00
Iter #     Field      Value
20         Best       1.065283e+00
Iter #     Field      Value
30         Best       1.065808e+00
Iter #     Field      Value
40         Best       1.065808e+00
Iter #     Field      Value
50         Best       1.065808e+00
The Best Evaluation: 1.065808e+00.
The Best String:
#    0: [       4], [       2], [       1], [       5], [       3], [       0]
#    6: [       7], [       6]

 best keff =  1.06416554234  and peak =  2.00346511469
Elapsed time =  108.014204025  seconds.
# Evaluations =  1990
***Destroying PGA context***
***Destroying PGA context***
```

Running it using `mpirun -np 5 python example10.py` yields the following output:

```
***Constructing PGA***
***Constructing PGA***
***Constructing PGA***
***Constructing PGA***
***Constructing PGA***
Iter #     Field      Value
10         Best       1.060239e+00
Iter #     Field      Value
20         Best       1.065283e+00
Iter #     Field      Value
30         Best       1.065808e+00
Iter #     Field      Value
40         Best       1.065808e+00
Iter #     Field      Value
50         Best       1.065808e+00
The Best Evaluation: 1.065808e+00.
The Best String:
#    0: [       4], [       2], [       1], [       5], [       3], [       0]
#    6: [       7], [       6]

 best keff =  1.06416554234  and peak =  2.00346511469
Elapsed time =  51.1656098366  seconds.
# Evaluations =  1990
***Destroying PGA context***
***Destroying PGA context***
***Destroying PGA context***
***Destroying PGA context***
***Destroying PGA context***
```

The reference solution for this objective is 1.06611227 , which was found by directly solving each of the ~80000 possible solutions. As we can observe, the GA does quite well. The parallel performance is also quite good, which makes sense as this problem has a comparitively expensive evaluation function. Note that for `np` above 2, PGApack uses the master process for communications, and hence `np=5` has just 4 compute processes.


Convergence of Objective
1.068
1.066
1.064
1.062
1.060
1.058
1.056
objective
0
10
20
30
40
50
60
generation
HTBX
solution

CHAPTER

THREE

METHODS

To be completed.

CHAPTER

FOUR

API REFERENCE

4.1 Introduction

This section provides a reference for all functions defined in the `pypgapack` module that are *extensions* of the basic PGAPack API. All PGAPack functions are contained in the `pypgapack.PGA` class. The PGAPack library is typically used as follows:

```
double evaluate(PGAContext *ctx, int p, int pop);
PGAContext *ctx;
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
PGASetUp(ctx);
PGARun(ctx, evaluate);
PGADestroy(ctx);
```

The `ctx` object is created explicitly by the user and then passed as the first argument to all subsequent function calls, with function names taking the form `PGAxxx`. For `pypgapack`, `ctx` is a *private* member of `PGA` created during construction, and all `PGA` members drop the *PGA* prefix and the initial `ctx` argument. So, for example,

```
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
PGASetUp(ctx);
```

in C/C++ becomes

```
obj = pypgapack.PGA(sys.argv, PGA.DATATYPE_BINARY, 100, PGA.MAXIMIZE)
obj.SetUp()
```

in Python. For all functions included in PGAPack, the user is directed to the pgapack documentation. What follows is a description of the few new methods added for `pypgapack` that make life in Python a bit easier.

4.2 pypgapack API

The easiest way to see what `pypgapack` offers is to do the following:

```
>>> import pypgapack as pga
>>> dir(pga)
['PGA', 'PGA_swigregister', '__builtins__', '__doc__', '__file__',
```

```
 '__name__', '__package__', '_newclass', '_object', '_pypgapack',
 '_swig_getattr', '_swig_property', '_swig_repr', '_swig_setattr',
 '_swig_setattr_nondynamic']
```

This command works with any Python module. Our interest is in the `PGA` class. We do the same for this:

```
>>> dir(pga.PGA)
['BinaryBuildDatatype', 'BinaryCopyString', 'BinaryCreateString',
 'BinaryDuplicate', 'BinaryHammingDistance', 'BinaryInitString',
 'BinaryMutation', 'BinaryOneptCrossover', 'BinaryPrint',
 'BinaryPrintString', 'BinaryTwoptCrossover', 'BinaryUniformCrossover',...
```

and find a really long list of class members, most of which are directly from PGAPack. In the following, we document only those not included in PGAPack, as use of the PGAPack functionality is covered above (i.e. drop the `ctx` argument and `PGA` prefix).

class `PGA`

PGA wrapper class.

`__init__`(*argv*, *datatype*, *n*, *direction*)

Construct the PGA context. This essentially wraps the PGACreate function, so see the PGAPack documentation.

Parameters

- **argv** – system argument
- **datatype** – allele datatype; can be `PGA.DATATYPE_XXX`, where `XXX` is `BINARY`, `INTEGER`, and so on.
- **n** – size of the unknown, i.e. number of alleles of type `datatype`
- **direction** – either `PGA.MAXIMIZE` or `PGA.MINIMIZE`

`GetIntegerChromosome`(*p*, *pop*)

Get direct access to the *p*-th integer chromosome string in population *pop* .

Parameters

- **p** – string index
- **pop** – population index

Returns string as numpy array of integers

`GetRealChromosome`(*p*, *pop*)

Get direct access to the *p*-th double chromosome string in population *pop*.

Parameters

- **p** – string index
- **pop** – population index

Returns string as numpy array of floats

`SetInitString`(*f*)

Set a function for initializing strings. The function `f` provided **must** have the signature `f(p,`

`pop)`, but should almost certainly be an inerited class member with the signature `f(self, p, pop)`. See PGAPack documentation for more about user functions.

Parameters **f** – Python function

See Also:

Example 4: User-defined String Initialization for an example on string initialization.

`SetCrossover`(*f*)

Set a function for the crossover operation. The function `f` provided **must** have the signature `f(a,b,c,d,e,f)`, but should almost certainly be an inerited class member with the signature `f(self,a,b,c,d,e,f)`. See PGAPack documentation for more about user functions.

Parameters **f** – Python function

See Also:

Example 5: User-defined Crossover Operator for an example on setting the crossover operator.

`SetMutation`(*f*)

Set a function for the mutation operator. The function `f` provided **must** have the signature `f(p, pop, prob)`, but should almost certainly be an inerited class member with the signature `f(self, p, pop, prob)`. See PGAPack documentation for more about user functions.

Parameters **f** – Python function

See Also:

Example 6: User-defined Mutation Operator for an example on setting the mutation operator.

`SetEndOfGen`(*f*)

Set a function for an operator to be performed at the end of each generation. The function `f` provided **must** have the signature `f(pop)`, but should almost certainly be an inerited class member with the signature `f(self, pop)`. Such an operator can be used to implement hill-climbing heuristics. See PGAPack documentation for more about user functions.

Parameters **f** – Python function

See Also:

Example 7: User-defined End of Generation Operator for an example on setting the an end of generation operator.

CHAPTER

FIVE

LICENSE

pypgapack itself is licensed under the MIT license, as follows:

```
Copyright (c) 2011 Jeremy Roberts

Permission is hereby granted, free of charge, to any
person obtaining a copy of this software and associated
documentation files (the "Software"), to deal in the
Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall
be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

PGAPack is under its own U-Chicago license that explicitly allows derivatives and redistribution:

```
COPYRIGHT

The following is a notice of limited availability of the code, and disclaimer
which must be included in the prologue of the code and in all source listings
of the code.

(C) COPYRIGHT 2008 University of Chicago

Permission is hereby granted to use, reproduce, prepare derivative works, and
to redistribute to others. This software was authored by:

D. Levine
Mathematics and Computer Science Division
```

```
Argonne National Laboratory Group

with programming assistance of participants in Argonne National
Laboratory's SERS program.

GOVERNMENT LICENSE

Portions of this material resulted from work developed under a
U.S. Government Contract and are subject to the following license: the
Government is granted for itself and others acting on its behalf a paid-up,
nonexclusive, irrevocable worldwide license in this computer software to
reproduce, prepare derivative works, and perform publicly and display
publicly.

DISCLAIMER

This computer code material was prepared, in part, as an account of work
sponsored by an agency of the United States Government. Neither the United
States, nor the University of Chicago, nor any of their employees, makes any
warranty express or implied, or assumes any legal liability or responsibility
for the accuracy, completeness, or usefulness of any information, apparatus,
product, or process disclosed, or represents that its use would not infringe
privately owned rights.
```

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX

poropy Documentation

poropy Documentation

Release 0.1.0

Jeremy Roberts

December 18, 2011

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH POROPY

1.1 Background

`poropy` is a set of simple tools related to optimization of incore fuel management. This includes loading pattern optimization, for which `poropy.coretools` provides several tools. For optimizing bundles (enrichment, gad content, etc.), `poropy.bundletools` will provide utilities.

1.2 Building poropy

underconstruction!

1.3 Next Steps

The user is encouraged to see the collection of *Examples*, which includes optimization of a simple core.

CHAPTER

TWO

EXAMPLES

All examples are located in the `poropy/examples` and the reference output for all examples is in `poropy/examples/output`.

2.1 Small Core Examples

The first two examples use a small benchmark core to demonstrate basic capabilities of `poropy.coretools`. The core is described in the theory documentation. A helper script is included to build the reactor in `poropy` and is listed here.

```
# examples/small_core.py -- small benchmark reactor

import numpy as np

from poropy.coretools import Reactor, Assembly, Reflector, Laban

def make_small_core() :
    """ This returns a Reactor object for the small benchmark.
    """

    # Problem
    # =======

    # This is a small 69 assembly core modeled as a quarter
    # core.  It has three equal batches composed of fresh,
    # once-, and twice-burned fuel.  The materials actually
    # correspond a 17x17, 4.25 w/o, 92 IFBA PWR assembly at
    # 0, 15, and 30 MWd/kg.  The assembly pitch is 20.5036,
    # so this benchmark is not using the data realistically.
    # The default pattern is a

    # Stencil
    # =======

    # First, define the stencil.  Fuel regions are positive
    # integers, where each value corresponds to a "region".
    # In an optimization sequence, the user can exclude regions
    # from the search space completely.  Here, the central
```

```
    # bundle is given a special value so that we can keep it fixed.
    # Note the whole map must be specified, even if certain locations
    # are to be fixed by rotational symmetry.  This makes it easier
    # for the user visually (and me w/r to implementation.
    stencil = np.array([[2, 1, 1, 1, 1, 0], \
                        [1, 1, 1, 1, 1, 0], \
                        [1, 1, 1, 1, 1, 0], \
                        [1, 1, 1, 1, 0, 0], \
                        [1, 1, 1, 0, 0, 0], \
                        [0, 0, 0, 0, 0, 0]] )
    # Note that 0 indicates reflector.  -1's can be placed as void,
    # but we skip that here.

    # Also, the regions are to be indexed in a more natural way than
    # is standard.  The stencil is indexed as would be any matrix.
    # Hence a fuel location [i,j] corresponds to the [i,j]th location
    # in the pattern (using 0-based indexing)  In other words, there is
    # no alpha-numeric scheme.

    # Initial Loading Pattern
    # =======================

    # The pattern is a 1-D array of fuel identifiers.  These
    # identifiers correspond to assemblies to be defined below.
    # Unlike above, fuel is not specified if it is constrained by
    # rotation symmetry.  The fuel indices as
    # listed correspond to the their location in stencil using
    # a row-major storage.  We work with 1-D data for simplicity.
    pattern = np.array([ 2, 0, 1, 0, 2,      \
                           1, 0, 1, 2,      \
                           0, 1, 0, 2,      \
                           1, 0, 1,         \
                           2, 2           ], dtype='i')


    # Assembly Definitions
    # ====================

    # Define the assemblies.  The pattern above has three unique
    # values, so we need to define 3 unique assemblies.

    # For simple BOC cycle analysis, it would
    # be sufficient to keep only unique assemblies and then map them to
    # core locations.  Once burnup is accounted for, assemblies become
    # unique, and having an individual assembly object for each physical
    # assembly makes more sense; that's what we do.  Note, the construction
    # used below assumes the unique assemblies are defined in an order
    # corresponding to their index in pattern.

    # Physical assemblies, as it were.
    assemblies = []

    # The unique types we have available.
```

```
    unique_assemblies = []

    # Assemblies are built with the following signature:
    #   ('type', enrichment, burnup, array([D1,D2,A1,A2,F1,F2,S12]))

    # Fresh
    unique_assemblies.append(Assembly('IFBA', 4.25, 0.0,  \
                                      np.array([1.4493e+00, 3.8070e-01, \
                                                9.9000e-03, 1.0420e-01, \
                                                7.9000e-03, 1.6920e-01, \
                                                1.5100e-02])))
    # Once burned
    unique_assemblies.append(Assembly('IFBA', 4.25, 15.0, \
                                      np.array([1.4479e+00, 3.7080e-01, \
                                                1.1000e-02, 1.2000e-01, \
                                                6.9000e-03, 1.7450e-01, \
                                                1.4800e-02])))
    # Twice burned
    unique_assemblies.append(Assembly('IFBA', 4.25, 30.0, \
                                      np.array([1.4494e+00, 3.6760e-01, \
                                                1.1500e-02, 1.1910e-01, \
                                                6.0000e-03, 1.6250e-01, \
                                                1.4700e-02])))

    # Loop through and assign assemblies to each fuel location in the pattern.
    for i in range(0, len(pattern)) :
        assemblies.append(unique_assemblies[pattern[i]])

    # Use the Biblis reflector material (the only one for now)
    reflector = Reflector('biblis')

    # Assembly dimension.
    width = 23.1226

    # Build the Reactor
    # =================

    return Reactor(stencil, pattern, assemblies, reflector, width, Laban())
```

2.1.1 Example 1: Manual Scoping

`poropy.coretools` has several function to aid scoping loading patterns by manually. This shows a few.

```
# examples/example01.py

import small_core

# Here, we'll investigate the small benchmark core.  First,
# build it using the premade script.
reactor = small_core.make_small_core()

# View all the diagnostics down the chain.
```

```
reactor.display()

# Evaluate the default pattern.  We can grab the eigenvalue
# and peaking as return values.
k, p = reactor.evaluate()
print "k = ",k," p = ",p
# Alternatively, we can use print_params to display current
# values of all optimization parameters.  Currently only
# keff and the max peaking are retained.
reactor.print_params()

# We can also print the power peaking.
reactor.print_peaking()

# With this, we can try optimizing by hand a bit.  Peaking
# occurs at (0, 1).  Printing the pattern helps visualize this.
reactor.print_pattern()

# Do a swap and evaluate.
reactor.swap([0,1],[0,2])
reactor.print_pattern()
reactor.evaluate()
reactor.print_params()
reactor.print_peaking()

# That's a significant peaking reduction with just a slight decrease
# in keff.  However, there is a better pattern.  For this keff,
# the tradeoff curve in the theory document suggests a peaking
# on the order of 1.75 or maybe less.  But we won't find out
# by hand...
```

Running it yields the following output:

```
 poropy - diagnostic output

REACTOR:
    thermal power : 1000 MWth
   electric power :  300 MWel
    CORE:
        pattern:     [12  0  6  1 15  8  2  9 13  4 10  5 16 11  3  7 14 17]
        ASSEMBLIES:
             assembly:  0
                model:  IFBA
           enrichment:  4.25
               burnup:  0.0
            constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
                 kinf:  1.29677543186
             assembly:  1
                model:  IFBA
           enrichment:  4.25
               burnup:  0.0
            constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
                 kinf:  1.29677543186
```

```
  assembly:  2
     model:  IFBA
enrichment:  4.25
    burnup:  0.0
 constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
      kinf:  1.29677543186
  assembly:  3
     model:  IFBA
enrichment:  4.25
    burnup:  0.0
 constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
      kinf:  1.29677543186
  assembly:  4
     model:  IFBA
enrichment:  4.25
    burnup:  0.0
 constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
      kinf:  1.29677543186
  assembly:  5
     model:  IFBA
enrichment:  4.25
    burnup:  0.0
 constants:  1.4493 0.3807 0.0099 0.1042 0.0079 0.1692 0.0151 0.025
      kinf:  1.29677543186
  assembly:  6
     model:  IFBA
enrichment:  4.25
    burnup:  15.0
 constants:  1.4479 0.3708 0.011 0.12 0.0069 0.1745 0.0148 0.0258
      kinf:  1.10161498708
  assembly:  7
     model:  IFBA
enrichment:  4.25
    burnup:  15.0
 constants:  1.4479 0.3708 0.011 0.12 0.0069 0.1745 0.0148 0.0258
      kinf:  1.10161498708
  assembly:  8
     model:  IFBA
enrichment:  4.25
    burnup:  15.0
 constants:  1.4479 0.3708 0.011 0.12 0.0069 0.1745 0.0148 0.0258
      kinf:  1.10161498708
  assembly:  9
     model:  IFBA
enrichment:  4.25
    burnup:  15.0
 constants:  1.4479 0.3708 0.011 0.12 0.0069 0.1745 0.0148 0.0258
      kinf:  1.10161498708
  assembly:  10
     model:  IFBA
enrichment:  4.25
    burnup:  15.0
 constants:  1.4479 0.3708 0.011 0.12 0.0069 0.1745 0.0148 0.0258
```

```
             kinf:  1.10161498708
         assembly:  11
            model:  IFBA
       enrichment:  4.25
           burnup:  15.0
        constants:  1.4479 0.3708 0.011 0.12 0.0069 0.1745 0.0148 0.0258
             kinf:  1.10161498708
         assembly:  12
            model:  IFBA
       enrichment:  4.25
           burnup:  30.0
        constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
             kinf:  0.994529582556
         assembly:  13
            model:  IFBA
       enrichment:  4.25
           burnup:  30.0
        constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
             kinf:  0.994529582556
         assembly:  14
            model:  IFBA
       enrichment:  4.25
           burnup:  30.0
        constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
             kinf:  0.994529582556
         assembly:  15
            model:  IFBA
       enrichment:  4.25
           burnup:  30.0
        constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
             kinf:  0.994529582556
         assembly:  16
            model:  IFBA
       enrichment:  4.25
           burnup:  30.0
        constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
             kinf:  0.994529582556
         assembly:  17
            model:  IFBA
       enrichment:  4.25
           burnup:  30.0
        constants:  1.4494 0.3676 0.0115 0.1191 0.006 0.1625 0.0147 0.0262
             kinf:  0.994529582556
    REFLECTOR:
            model:  biblis
        constants:  1.32 0.2772 0.0026562 0.071596 0.0 0.0 0.023106 0.0257622
EVALUATOR:  LABAN-PEL
    input:  laban0.inp
   output:  laban0.out
k =  1.137119  p =  2.0687

 optimization parameters
 -----------------------
```

```
    keff    =  1.137119
 maxpeak    =  2.0687

[[ 1.64865  2.0687   1.53543  1.18246  0.40744  0.     ]
 [ 2.0687   1.7359   1.69436  0.94983  0.34894  0.     ]
 [ 1.53543  1.69436  1.19476  0.8788   0.26544  0.     ]
 [ 1.18246  0.94983  0.8788   0.43841  0.       0.     ]
 [ 0.40744  0.34894  0.26544  0.       0.       0.     ]
 [ 0.       0.       0.       0.       0.       0.     ]]

 current loading pattern
 -----------------------
       0   1   2   3   4
  ----------------------
 0|   12   0   6   1  15
 1|   rs   8   2   9  13
 2|   rs   4  10   5  16
 3|   rs  11   3   7 ref
 4|   rs  14  17 ref ref


 current loading pattern
 -----------------------
       0   1   2   3   4
  ----------------------
 0|   12   6   0   1  15
 1|   rs   8   2   9  13
 2|   rs   4  10   5  16
 3|   rs  11   3   7 ref
 4|   rs  14  17 ref ref


 optimization parameters
 -----------------------
    keff    =  1.135264
 maxpeak    =  1.85174

[[ 1.22885  1.47989  1.85174  1.31604  0.4458   0.     ]
 [ 1.47989  1.53051  1.73407  1.0157   0.37678  0.     ]
 [ 1.85174  1.73407  1.21558  0.91612  0.2809   0.     ]
 [ 1.31604  1.0157   0.91612  0.45609  0.       0.     ]
 [ 0.4458   0.37678  0.2809   0.       0.       0.     ]
 [ 0.       0.       0.       0.       0.       0.     ]]
```

2.1.2 Example 2: Optimizing a Core

For this example, the optimization tools in `poropy.coretools.optimizer` are demonstrated. Many parameters are specific to pypgapack.

```
1 # examples/example02.py
2
3 import small_core
```

```
4
5  from poropy.coretools import Optimizer
6  from pypgapack import PGA
7  from mpi4py import MPI
8  import numpy as np
9  import sys
10 import matplotlib.pyplot as plt
11 from matplotlib import rc
12 rc('text', usetex=True)
13 rc('font', family='serif')
14
15 # Optimizer
16 # =========
17 #
18 # While Optimizer has some default definitions,
19 # we make an inherited class to set our own
20 # objective function.  Moreover, because we are
21 # fixing the central bundle to be twice burned,
22 # we need special initialization and mutation
23 # functions.
24
25 class OptimizeSmallCore(Optimizer) :
26     """  Derive our own class from PGA.
27     """
28     def small_core_objective(self, p, pop) :
29         """ Minimize peaking and maximize keff using weighted objective.
30
31         Here, we seek to minimize p for k >= 1.135. Note that p=1.85 is
32         about what we found manually, albeit with just one swap from the
33         default pattern.
34         """
35         pattern = self.GetIntegerChromosome(p, pop)
36         self.reactor.shuffle(pattern)
37         keff, peak = self.reactor.evaluate()
38         val = self.fun(keff,peak)
39         self.evals += 1
40         #print val, keff, peak, pattern
41         del pattern
42         return val
43
44     def fun(self,k,p):
45         delta = 0
46         if k < 1.135 :
47             delta = k - 1.135
48         return 1.0 * (1.85 - p) + 50.0 * delta
49
50
51     def swap(self, p, pop, pm) :
52         """  Random swap of bundles.
53
54         This example allows swapping for all but the central
55         element.  Moreover, no checking is done to ensure swaps
56         are not done between identical or forbidden bundles.  In
```

```
57          those cases, a swap simply doesn't happen.  This is one
58          area where a lot more work can be done w/r to implementation.
59          """
60          #pass
61          n = self.GetStringLength()
62          pattern = self.GetIntegerChromosome(p, pop)
63          # index is a random permutation of [1,2,... numberbundles-1]
64          #   i.e. zero is excluded (and zero is the central bundle)
65          index = np.random.permutation(n-1) + 1
66          i1 = index[0]
67          i2 = index[1]
68          tmp1 = pattern[i1]
69          tmp2 = pattern[i2]
70          pattern[i1] = tmp2
71          pattern[i2] = tmp1
72          del pattern
73          return 1 # A positive value means something swapped.
74
75      def init(self, p, pop) :
76          """  Random initial states.
77          """
78          n = self.GetStringLength()
79          pattern = self.GetIntegerChromosome(p, pop)
80          # perm is a random permutation of [0,1,... numberbundles-2]
81          #   i.e. 17 is excluded.  The 17th bundle is by definition
82          #   the least reactive, and so that be the default central.
83          perm = np.random.permutation(n-1)
84          pattern[0] = 17
85          for i in range(1, n) :
86              pattern[i] = perm[i-1]
87          #print " pattern=", pattern
88          del pattern
89
90  # Begin Optimization
91  # ==================
92
93  # MPI things.  These should be commented out if
94  # a serial version of pypgapack is used.
95  comm = MPI.COMM_WORLD   # Get the communicator.
96  rank = comm.Get_rank()  # Get my rank.
97  time = MPI.Wtime()      # Start the clock.
98
99  # Get the small reactor.
100 reactor = small_core.make_small_core()
101
102 vals = np.zeros(10)
103 kefs = np.zeros(10)
104 peks = np.zeros(10)
105 evas = np.zeros(10)
106
107 for i in range(0, 1) : # just one run for example output
108     # Create an optimizer.
109     opt = OptimizeSmallCore(sys.argv, reactor)
```

```
110
111     # Set initialization and swapping functions.
112     opt.SetInitString(opt.init)
113     opt.SetMutation(opt.swap)
114
115     # Set various PGA parameters 2
116     opt.SetNumReplaceValue(10)
117
118     opt.SetRandomSeed(i+1)    # Set random seed for verification.  # VERIFY, seed=80,1 pop=
119     np.random.seed(i+1)       # Do the same with Numpy.
120     opt.SetPopSize(50)           # Large enough to see some success.
121     opt.SetMaxGAIterValue(500)   # Small number for output.
122
123     # Set various Optimizer parameters
124     opt.set_track_best(True)     # Track best everything each generation
125     opt.set_fixed_central(True)  # Fix the central bundle.
126
127     # Run the optimization.  This implicitly performs both SetUp
128     #   and Run of PGA.
129     opt.run(opt.small_core_objective)
130
131     if rank == 0 :
132         best = opt.GetBestIndex(PGA.OLDPOP)     # Get the best string
133         bestpattern = opt.GetIntegerChromosome(best, PGA.OLDPOP)
134         print " best pattern ", bestpattern
135         reactor.shuffle(bestpattern)
136         reactor.evaluate()
137         reactor.print_params()
138         reactor.print_pattern()
139         reactor.print_peaking()
140         vals[i]=opt.fun(reactor.evaluator.keff, reactor.evaluator.maxpeak)
141         kefs[i]=reactor.evaluator.keff
142         peks[i]=reactor.evaluator.maxpeak
143         evas[i]=opt.evals
144 if rank == 0 :
145     print np.mean(vals), np.std(vals)
146     print np.mean(kefs), np.std(kefs)
147     print np.mean(peks), np.std(peks)
148     print np.mean(evas), np.std(evas)
149
150
151 opt.Destroy()  # Clean up PGAPack internals.
```

Running it yields the following output:

```
***Constructing PGA***
Iter #     Field      Value
10         Best       -1.600000e-01
Iter #     Field      Value
20         Best       -1.253100e-01
Iter #     Field      Value
30         Best       -1.253100e-01
Iter #     Field      Value
40         Best       -1.253100e-01
```

```
Iter #     Field      Value
50         Best       -1.145000e-01
Iter #     Field      Value
60         Best       -1.135300e-01
Iter #     Field      Value
70         Best       -7.820000e-03
Iter #     Field      Value
80         Best       2.336000e-02
Iter #     Field      Value
90         Best       2.336000e-02
Iter #     Field      Value
100        Best       2.336000e-02
Iter #     Field      Value
110        Best       2.336000e-02
Iter #     Field      Value
120        Best       4.128000e-02
Iter #     Field      Value
130        Best       4.128000e-02
Iter #     Field      Value
140        Best       4.128000e-02
Iter #     Field      Value
150        Best       4.128000e-02
Iter #     Field      Value
160        Best       4.128000e-02
Iter #     Field      Value
170        Best       4.128000e-02
Iter #     Field      Value
180        Best       4.128000e-02
Iter #     Field      Value
190        Best       4.128000e-02
Iter #     Field      Value
200        Best       4.128000e-02
Iter #     Field      Value
210        Best       4.128000e-02
Iter #     Field      Value
220        Best       4.128000e-02
Iter #     Field      Value
230        Best       4.128000e-02
Iter #     Field      Value
240        Best       4.128000e-02
Iter #     Field      Value
250        Best       4.128000e-02
Iter #     Field      Value
260        Best       4.128000e-02
Iter #     Field      Value
270        Best       5.823000e-02
Iter #     Field      Value
280        Best       5.823000e-02
Iter #     Field      Value
290        Best       5.823000e-02
Iter #     Field      Value
300        Best       5.823000e-02
Iter #     Field      Value
```

```
310         Best        5.823000e-02
Iter #      Field       Value
320         Best        5.823000e-02
Iter #      Field       Value
330         Best        5.823000e-02
Iter #      Field       Value
340         Best        5.823000e-02
Iter #      Field       Value
350         Best        5.823000e-02
Iter #      Field       Value
360         Best        5.823000e-02
Iter #      Field       Value
370         Best        5.823000e-02
Iter #      Field       Value
380         Best        5.823000e-02
Iter #      Field       Value
390         Best        5.823000e-02
Iter #      Field       Value
400         Best        5.823000e-02
Iter #      Field       Value
410         Best        5.823000e-02
Iter #      Field       Value
420         Best        5.823000e-02
Iter #      Field       Value
430         Best        1.027200e-01
Iter #      Field       Value
440         Best        1.027200e-01
Iter #      Field       Value
450         Best        1.106000e-01
Iter #      Field       Value
460         Best        1.174900e-01
Iter #      Field       Value
470         Best        1.174900e-01
Iter #      Field       Value
480         Best        1.174900e-01
Iter #      Field       Value
490         Best        1.174900e-01
Iter #      Field       Value
500         Best        1.174900e-01
The Best Evaluation: 1.174900e-01.
The Best String:
#    0: [      17], [      11], [       6], [       5], [      10], [       7]
#    6: [       2], [       4], [      13], [       3], [       1], [       8]
#   12: [      14], [       0], [       9], [      12], [      15], [      16]

 best pattern  [17 11  6  5 10  7  2  4 13  3  1  8 14  0  9 12 15 16]

 optimization parameters
 -----------------------
    keff   =  1.140122
 maxpeak   =  1.73251

 current loading pattern
```

```
 ----------------------
       0   1   2   3   4
  ---------------------
 0|   17  11   6   5  10
 1|   rs   7   2   4  13
 2|   rs   3   1   8  14
 3|   rs   0   9  12 ref
 4|   rs  15  16 ref ref


[[ 1.07658  1.28113  1.4042   1.37252  0.58032  0.     ]
 [ 1.28113  1.4031   1.73251  1.30587  0.43526  0.     ]
 [ 1.4042   1.73251  1.53302  0.78678  0.27084  0.     ]
 [ 1.37252  1.30587  0.78678  0.34406  0.       0.     ]
 [ 0.58032  0.43526  0.27084  0.       0.       0.     ]
 [ 0.       0.       0.       0.       0.       0.     ]]
0.011749 0.035247
0.1140122 0.3420366
0.173251 0.519753
506.0 1518.0
***Destroying PGA context***
```

CHAPTER

THREE

API REFERENCE MANUAL

This chapter descripes the application programming interface for `poropy`. Currently, `poropy` has two packages, `coretools` and `bundletools`. The former is included here for now while the latter is swimming in infancy.

3.1 `poropy.coretools` — The coretools package

`poropy.coretools` aims to provide a complete set of tools for analyzing and optimizing loading patterns for a simple set of inputs. Currently, it contains a number of modules, each with one or more classes. The current documentation is quite basic, as the interfaces and such are in a state of flux.

3.2 `coretools.reactor` — The reactor module

class `reactor.`**`Core`**(*stencil*, *pattern*, *assemblies*, *reflector*, *width*)

Represents the core.

`display`()

Print my information.

`inventory_size`()

Return the inventory size.

`make_fuel_map`()

Map the fuel to (i,j) for easy swapping.

`print_pattern`()

Print the pattern in a nice 2-D format.

`sort_assemblies`(*pattern*, *assemblies*)

Sort the assemblies by reactivity.

`swap`(*x*, *y*)

Swap two bundles.

`update_pattern`(*pattern*)

Update the pattern.

class reactor.**Reactor**(*stencil*, *pattern*, *assemblies*, *reflector*, *width*, *evaluator*)
Represents the reactor

display()
Print out all information (for debugging).

evaluate()
Evaluate the current core. Pattern must be up-to-date.

number_bundles()
Return the number of fuel bundles.

print_params()
Print out optimization parameters.

print_pattern()
Print out the peaking factor matrix.

print_peaking()
Print out the peaking factor matrix.

shuffle(*pattern*)
Update the pattern.

This applies *only* to within-core shuffling. Accounting for different new fuel or swaps with the pool will need something more to handle the assemblies issue.

swap(*x*, *y*)
Swap two bundles.

class reactor.**SpentFuelPool**
Represents the spent fuel pool. *NOT IMPLEMENTED*

display()
Display my contents.

3.3 coretools.assembly — The assembly module

Created on Dec 14, 2011

@author: robertsj

class assembly.**Assembly**(*model='IFBA', enrichment=4.0, burnup=1.0, data=[1.4493, 0.3807, 0.0099, 0.1042, 0.0079, 0.1692, 0.0151]*)
Represents a fuel assembly.

burn(*burnup*)
Stub method for inserting a data model.

display()
Display my contents.

get_constants()
Explicitly change the group constants.

get_constants_list()
Explicitly change the group constants.

This version returns a list, which might be convenient.

kinf()
Return kinf.

set_constants(*data*)
Explicitly change the group constants.

class assembly.**Reflector**(*model*)
Represents a reflector.

burn(*burnup*)
Stub method for inserting a data model.

display()
Display my contents.

class assembly.**Void**
Represents a reflector.

assembly.**assembly_compare**(*x*, *y*)
Compare assemblies based on BOC kinf; sorted by descending reactivity.

3.4 coretools.assembly — The laban module

Created on Dec 11, 2011

@author: robertsj

class laban.**Laban**(*rank=0*, *order=0*, *tolerance=0.001*)
Uses the LABAN-PEL code to evaluate loading patterns.

display()
Introduce myself.

evaluate()
Evaluate the current core.

make_input_map()
Print the map of assemblies.

make_input_materials()
Print the materials definitions.

make_input_top()
Create the string for the top invariant part of the input.

plot_peak()
Plot the power peaking factors.

read()
Read a LABAN-PEL output file and load various data.

run()
Run LABAN-PEL (must set input first)

setup(*core*)
Set up some structures needed before solving.

3.5 coretools.assembly — The vendor module

class vendor.**Vendor**
Fresh fuel vendor. *NOT IMPLEMENTED*

3.6 coretools.assembly — The optimizer module

class optimizer.**Optimizer**(*argv*, *reactor*)
Derive our own class from PGA.

eog()
Do something at the end of each generation.

In general, this a very customizable routine. This is where hill-climbing heuristics can be placed. Additionally, tracking of objectives as a function of generation is easily done. For this default implementation, the best keff and maxpeak are kept for each generation.

htbx(*p1*, *p2*, *pop1*, *c1*, *c2*, *pop2*)
Heuristic tie-breaking cross-over.

Note, this implementation *assumes* that the assemblies are sorted by reactivity *before* optimization. The only geometric constraint considered is a fixed central bundle, and admittedly, the treatment isn't as clean as it could be. Future work...

objective(*p*, *pop*)
Minimize peaking and maximize keff using weighted objective.

The default seeks to maximize k for p < 1.55.

run(*f*)
Optimize the reactor for the objective f

set_fixed_central(*value=True*)
Fix the central bundle.

By construction, the central bundle is the first entry in the pattern. This flag will indicate crossover should skip this.

set_track_best(*value=True*)
Track the best evaluations.

CHAPTER

FOUR

LICENSE

poropy is licensed under the MIT license, as follows:


```
Copyright (c) 2011 Jeremy Roberts

Permission is hereby granted, free of charge, to any
person obtaining a copy of this software and associated
documentation files (the "Software"), to deal in the
Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall
be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

U

V