User Guide for Reduced-Hamiltonian Taylor Coefficient Generation

Jake Smith
jsmith74@tulane.edu

August 4, 2016

Contents

1 Compiling the C++ Code

Before we begin, we need to compile some C++ code. A makefile is included with the source files, that will automatically build a binary executable and call it *RTCGen*. Some non-standard libraries are required:

- libuuid
- Eigen3
- WSTP

Eigen3 and WSTP should be included with the source code files pulled from the TeamForge git repository, and require no additional installation. If you want to compile this program on an OS other than Linux, you will need to replace the WSTP library with the version included with your installation of *Mathematica*. Uuid is available through standard package repositories- yum on CentOS, or apt-get on Ubuntu.

The user should never need to recompile any code; *RTCGen* was written to be completely general. All physical and algorithm control parameters are input via a *Mathematica* notebook.

2 Running the Code and Reading the Output

2.1 A Quick How-to-Use Guide

This purpose of this program is to approximate a large user-defined Hamiltonian in some small dimensional Hilbert space, and to Taylor expand this result. This will allow an efficiently repeatable after-the-fact consideration of noise in a physical simulation.

For the purpose of usability, the *RTCGen* was written to allow the user to input all physical definitions in a *Mathematica* notebook. These definitions are:

- The eigen decomposition of a "large" Hamiltonian in some initial basis as a function of a set of flux parameters
- A predefined, time-dependent ideal (free of noise) path through flux space

In order to define the eigen decomposition of the Hamiltonian, we write a function in *Mathematica* called **es**, which takes in a vector position in flux space and returns a (possibly truncated) list of eigenvalues and eigenvectors of our full Hamiltonian. E.g. for the prototypical two level system with flux parameters ε and Δ,

$$H(\varepsilon(t), \Delta(t)) = \frac{1}{2}\begin{bmatrix} \varepsilon(t) & \Delta(t) \\ \Delta(t) & -\varepsilon(t) \end{bmatrix} \tag{1}$$

we can write the *Mathematica* function

$$\texttt{es[}\mathit{FLUXVALS\_}\texttt{] := Eigensystem}\left[0.5\begin{pmatrix} \mathit{FLUXVALS}\texttt{[[1]]} & \mathit{FLUXVALS}\texttt{[[2]]} \\ \mathit{FLUXVALS}\texttt{[[2]]} & -\mathit{FLUXVALS}\texttt{[[1]]} \end{pmatrix}\right]\texttt{;} \tag{2}$$

The output of **es** must be a *Mathematica* List of two elements; the first being a List of eigenvalues, and the second being a List of corresponding eigenvectors. The eigensystem must be order in order of increasing eigenvalues, and all eigenvectors must be normalized. This is usually the format of the output of *Mathematica*'s built in eigen-solver, **Eigensystem**, and is the format of the realistic flux qubit eigen-decomposition Jamie provided. Occasionally, **Eigensystem** will not output in this format by default - the eigenvalues will not be ordered or the eigenvectors will not be normalized. You may need to add some extra code to fix this.

In order to define the ideal path through flux space, we write a function **fluxVec**, which takes in a time and returns a vector of instantaneous flux values. E.g. for our Hamiltonian above, say we want to follow the ideal flux path:

$$\varepsilon(t) = \frac{1}{4}\cos(2.0\pi t) \quad \Delta(t) = 11.0 \tag{3}$$

then we write a *Mathematica* function,

```
fluxVec[t_] := {0.25 Cos[2.0 Pi t], 11.0};
```
(4)

The output of **fluxVec** must be a List of Real numbers representing a position in flux space.

Next, the user can choose the output file directory; the location where the Taylor coefficients and any debugging files will be saved by using the standard **SetDirectory** function:

```
SetDirectory["/home/ja26765/"];
```
(5)

Now, we need to link our notebook to the compiled Taylor coefficient generator program. This is done with *Mathematica*'s **Install** function. By default, I've named our compiled program *RTCGen* in the makefile, so the proper function call would be (assuming the binary was in my home directory)

```
Install["/home/ja26765/RTCGen"]
```
(6)

Finally, the user needs to tell *Mathematica* to generate the taylor coefficients of the reduced-Hamiltonian. This is done by calling the function **GenerateTaylorCoefficients**, with the following arguments:

```
nlevsInitial = 2;
nlevs = 2;
numbFluxParameters = 2;
tInitial = 0.0;
tFinal = 0.01;
taylorExpansionOrder = 3;


totalPoints = 10;
RDagSamplingPoints = 5;
dHdphiSamplingPoints = 3;
dHdphiFiniteDifferenceSize = 0.5;

GenerateTaylorCoefficients[nlevsInitial, nlevs, numbFluxParameters, tInitial,
 tFinal, taylorExpansionOrder, totalPoints, RDagSamplingPoints, dHdphiSamplingPoints,
 dHdphiFiniteDifferenceSize]
```

A quick rundown of these arguments:

- **nlevsInitial** is the Hilbert space dimension of the large Hamiltonian in the initial energy eigenbasis.

- **nlevs** is the reduced Hilbert space dimension of our truncated system
- **numbFluxParameters** is the dimension of the flux space
- **tInitial** is the starting time for our journey through flux space
- **tFinal** is the ending time for our journey through flux space
- **taylorExpansionOrder** is the highest order set of coefficients in the taylor expansion that we want the program to evaluate

The remaining four arguments; **totalPoints, RDagSamplingPoints, dHdphiSamplingPoints** and **dHdphiFiniteDifferenceSize** are non-physical parameters that we will go into more detail in section 3.

For now, you should be able to at least *run* this code for any physical system you choose. If successful, running the function **GenerateTaylorCoefficients** should return the following message:

```
Successfully exported Reduced Hamiltonian Taylor Coefficients to file.
```
(7)

and the files "*RTC.dat*", "*evH0.dat*", "*Taylor_Coefficient_Order.dat*" and "*finalEb.dat*" should appear in the directory specified. If this message does not appear, and the output files containing the taylor coefficients are nowhere to be found, the reason is likely due to *Mathematica*'s way of implicitly defining variable types; see section 2.2.

As a final step, you can unlink my program from your *Mathematica* notebook if you want.

```
Uninstall["/home/ja26765/nonAdiab"];
```
(8)

Neat.

Included with the source files and this PDF is an example notebook, *Example Mathematica Notebook.nb* which defines **es** for the more complicated, realistic flux qubit provided by Jamie, and an arbitrarily chosen pulse signal for **fluxVec**.

2.2 Data Types

Because we're exchanging data between a compiled C++ binary and a *Mathematica* notebook, the user needs to be at least a little bit aware of how data types are declared in both languages.

In C++, all variables must be declared with an explicit type (e.g. int, double, long, char, etc).

In *Mathematica* all data types are declared implicitly. There are only two numeric types; **Integer** and **Real**, which are analogous to C++'s **int** and **double**, respectively. The type is assigned depending on if the user included a decimal point in their definition or not:

nlevsInitial	Integer
nlevs	Integer
numbFluxParameters	Integer
tInitial	Real
tFinal	Real
taylorExpansionOrder	Integer
totalPoints	Integer
RDagSamplingPoints	Integer (must be odd)
dHdphiSamplingPoints	Integer (must be odd)
dHdtFiniteDifferenceSize	Real

Figure 1: Parameters and Their Implicitly defined Datatypes

C++	*Mathematica*
int x = 5;	x = 5;
double x = 5.0;	x = 5.0;

Calling **GenerateTaylorCoefficients** in a *Mathematica* notebook will not work unless all of the input parameters are of the correct data type, **Integer** or **Real**. For a list of the explicit type definitions, refer to Fig. (1).

2.3 Format of the Output

Running the code successfully will create four data files in the directory specified by the user. These files are

- *evH0.dat*
- *RTC.dat*
- *Taylor_Coefficient_Order.dat*
- *finalEb.dat*

evH0.dat contains a series of columns separated by a tabulator character. The first column contains a list of time values; these are the points in time at which you are diagonalizing your "large" Hamiltonian. The remaining columns will be the **nlevs** lowest eigenvalues at each time (at least the lowest at **tInit** - the code should correct any degenerate crossing). *evH0.dat* is generated as a way for the user to verify that everything went alright - plot these values and make sure any crossings went well, and that the resolution is appropriate, if you so desire.

RTC.dat contains the coefficients for the taylor expansion of our reduced Hamiltonian. This file should be readable to humans in the current version of the code. First, a double precision number gives the starting time **tInit**. Then , there should be a set of (**nlevs** by **nlevs**) complex matrices. These are the taylor expansion coefficients. Then, some extra line breaks, and then the succeeding time value and its corresponding taylor coefficients. This cycle continues over the duration of the time interval specified by the user in the *Mathematica* code. I'll give a short description of what these matrices are in the next section, and then how to use them in the section after that. The format of how these matrices are exported was an arbitrary decision. If you want to change the I/O style, you can edit the commented function at the top of the file *reducedTaylorCofficientGen.cpp*, or send me an email and I'd be happy to change it.

2.4 Multivariable Taylor Expansion

The main idea here is that we're expanding our flux-dependent "small" Hamiltonian in the instantaneous eigenbasis around ideal points in flux space, to allow noise-perturbations to be considered later. Formally, lets define

$\vec{\phi}(t) = \{\phi_1(t), \phi_2(t) ... \phi_m(t)\}$ as an ideal (noiseless) position in m-dimensional flux space at time t. This vector is determined by our *Mathematica* function, **fluxVec**.

$\vec{x}(t) = \{x_1(t), x_2(t) ... x_m(t)\}$ as some perturbative vector that represents our flux noise at time t. This "noise position" will be incorporated later, during an actual simulation.

With noise, our position in flux space is therefore $\vec{\phi}(t) + \vec{x}(t)$, and our exact Hamiltonian in the instantaneous eigenbasis at time t is determined by

$$H(\vec{\phi}(t) + \vec{x}(t)) \tag{9}$$

We parametrize the length of the noise vector and apply taylor's theorem:

$$G(q) = H(\vec{\phi}(t) + q\vec{x}(t)) \tag{10}$$

$$H(\vec{\phi}(t) + \vec{x}(t)) = G(1) = G(0) + G'(q)|_{q=0} + \frac{1}{2}G''(q)|_{q=0} + ... \tag{11}$$

Applying the chain rule, we find

$$H(\vec{\phi}(t) + \vec{x}(t)) = \sum_{n=0} \frac{F_n(\vec{x}(t))}{n!} \tag{12}$$

where

$$F_n(\vec{x}(t)) = \sum_{r_1, r_2, ..., r_n = 1}^{m} \frac{\partial^n H(\vec{\phi}(t) + \vec{\xi}(t))}{\partial\xi_{r_1}\partial\xi_{r_2} ... \partial\xi_{r_n}}|_{\vec{\xi}=\vec{0}} \cdot x_{r_1}(t) \cdot x_{r_2}(t) \cdot ... \cdot x_{r_n}(t) \tag{13}$$

When running an actual physical simulation, the user must define some time dependent flux-noise function, $x(t)$, and evolve the system with the Hamiltonian,

$$H_{noise}(\vec{x}(t)) = \sum_{n=0}^{N} \frac{F_n(\vec{x}(t))}{n!} \tag{14}$$

over each discrete time interval, where N is the expansion order specified by the user.

The matrices contained in the file *RTC.dat* are the coefficients

$$\boxed{C_t(n,\vec{r}) = \frac{\partial^n H(\vec{\phi}(t)+\vec{\xi}(t))}{\partial\xi_{r_1}\partial\xi_{r_2}...\partial\xi_{r_n}}|_{\vec{\xi}=\vec{0}}} \tag{15}$$

which we have indexed with a vector labeling $\vec{r}$ and expansion order n. Thus, the instantaneous Hamiltonian at time t, as a function of flux noise is

$$\boxed{H_{noise}(\vec{x}(t)) = \sum_{n,\vec{r}} \frac{C_t(n,\vec{r})}{n!} \cdot x_{r_1}(t) \cdot x_{r_2}(t) \cdot ... \cdot x_{r_n}(t)} \tag{16}$$

where $n : 0 \rightarrow N$ and $\vec{r}$ is over all distinct partial derivatives of the m flux variables.

The total number of coefficient matrices, $C_n(t,\vec{r})$, generated at each point in time depends on the dimension of the flux space, m, and the expansion order of our taylor series, N:

$$\text{Number of } C_t(n,\vec{r}) \text{ matrices } = \sum_{n=0}^{N} \frac{(n+m-1)!}{n!(m-1)!} \tag{17}$$

Deriving eq. (17) is a neat little exercise.

The $C_t(n,\vec{r})$ at each time t are printed in *RTC.dat* in a hierarchical order. At the highest priority, the coefficients are in order of taylor expansion order, n. Below that, we use reverse lexicographic ordering of the $\vec{r}$ vectors. We can flatten this total ordering into a single index, which I'll just call l. As an example, refer to Fig. (2) the ordering for a system in 2-dimensional flux space expanded to 3rd order.

l	n	$\vec{r}$
1	0	()
2	1	(1)
3	1	(2)
4	2	(1,1)
5	2	(1,2)
6	2	(2,2)
7	3	(1,1,1)
8	3	(1,1,2)
9	3	(1,2,2)
10	3	(2,2,2)

Figure 2: Ordering of the $C_t(n,\vec{r})$ Matrices as they are printed in *RTC.dat* for $m = 2$ and $N = 3$

2.5 Direct step-by-step Instructions for using *H0.dat* to build the Hamiltonians with noise in your simulation

The following are the steps for building the approximation to the Hamiltonian with flux-noise, $H_{noise}(\vec{x}(t))$ at each time, t:

First, import the complex matrices in the order they are saved in *RTC.dat* and index them in order with a single integer l. Call these matrices $C_t(l)$.

Open *Taylor_Coefficient_Order.dat*. Each line in this file is a vector of integers similar to the $\vec{r}$ vector of the previous section. There should be $\sum_{n=0}^{N} \frac{(n+m-1)!}{n!(m-1)!}$ vectors where each vector is of length m. Import them into memory in the order they are provided. Again, mark each of these vectors in order with the same index, l, and call them $\vec{v}(l)$.

The third and final component needed to build the Hamiltonian is the instantaneous flux-noise, $\vec{x}(t)$. This vector function is to be provided by the user during the physical simulation.

Putting all three components together, we should have our approximate Hamiltonian, ready for use in a physical simulation over some small time interval with midpoint at t;

$$\boxed{H_{noise}(\vec{x}(t)) = \sum_{l} \frac{C_t(l)}{(\sum_{i=1}^{m} v_i(l))!} \cdot x_1(t)^{v_1(l)} \cdot x_2(t)^{v_2(l)} \cdot \ldots \cdot x_m(t)^{v_m(l)}} \tag{18}$$

$v_i(l)$ is the ith component of $\vec{v}(l)$, $x_j(t)$ is the jth component of $\vec{x}(t)$, m is the flux space dimension and

$$l : 1 \rightarrow \sum_{n=0}^{N} \frac{(n+m-1)!}{n!(m-1)!} \tag{19}$$

That's it; you should now be able to use *RTCGen* for any physical system, and interpret the results.

The remainder of this document will be on the significance of the four non-physical parameters in the code, **totalPoints, RDagSamplingPoints, dHdphiSamplingPoints** and **dHdphiFiniteDifferenceSize**.

3 Setting the Four Non-Physical Parameters

3.1 totalPoints and RDagSamplingPoints

The total number of points that the user wants to sample along the ideal path through flux space is set by **totalPoints**. The user should choose a high enough number that accurately captures the details in the evolution of the eigenvalues and eigenvectors of the matrix in the inital eigenbasis.

The Hamiltonian in the instantaneous eigenbasis at a particular position in flux space, $\phi(t)$, is given by

$$H(\phi(t)) = R(\phi(t))H_0(\phi(t))R^\dagger(\phi(t)) - \frac{i}{2\pi}R(\phi(t))\frac{dR^\dagger(\phi(t))}{dt} \tag{20}$$

$R^\dagger(\phi(t))$ is an (**nlevsInitial** by **nlevs**) dimensional matrix where the columns are the eigenvectors corresponding to the **nlevs** lowest eigenvalues of the Hamiltonian in the initial eigenbasis, $H_0(\phi(t))$. $\frac{dR^\dagger(\phi(t))}{dt}$ is evaluated by sampling an odd number of $R^\dagger(\phi(t))$ matrices along

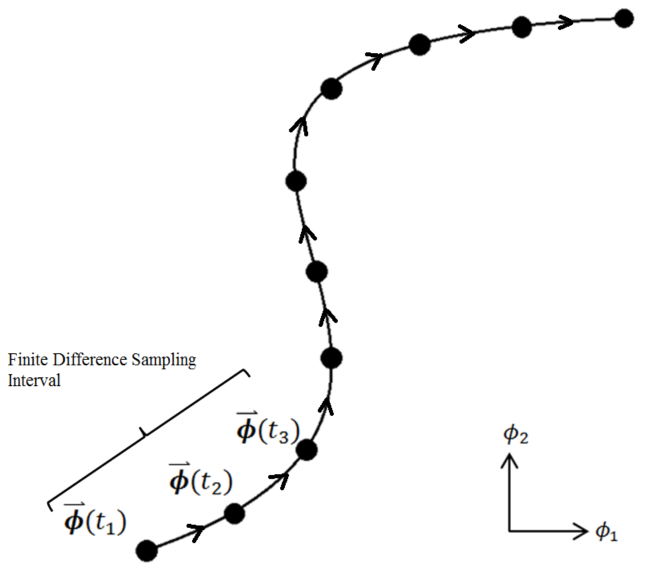

Figure 3: A Path in 2D Flux space, where **totalPoints**=10 and **RDagSamplingPoints**=3

the ideal path in flux space, and applying a central-finite difference formula.

The number of $R^{\dagger}(\phi(t))$ matrices the user wants to sample in finite difference formula is set by the parameter **RDagSamplingPoints**. Because the program uses a central-difference numerical derivative, this number must be odd. The finite difference coefficients are automatically generated using Fornberg's algorithm[1].

totalPoints and **RDagSamplingPoints** are closely related; the spacing of points in the finite difference evaluation of $\frac{dR^{\dagger}(\phi(t))}{dt}$ is going to be the same as total spacing determined by the resolution set in **totalPoints**. See Fig. (3.1). This restriction is made to significantly improve code efficiency (by several orders of magnitude, in fact). Instead of re-evaluating the eigensystem of $H_0(\phi(t))$ at multiple points along the flux-path, we instead store **RDagSamplingPoints** number of $R^{\dagger}(\phi(t))$ matrices in memory and replace the "oldest" eigensystem with the "newest" as we travel along the path in flux space. Note that increasing **RDagSamplingPoints** will therefore not significantly increase the computational workload of the program. It will, however, demand more memory usage.

In conclusion, the user should first choose **totalPoints** such that the resolution of the system accurately captures the evolution of the eigensystem of $H_0(\phi(t))$ along the flux path. They should then choose an **RdagSamplingPoints** such that the error in the finite difference formula is minimized for the point spacing established by **totalPoints.**

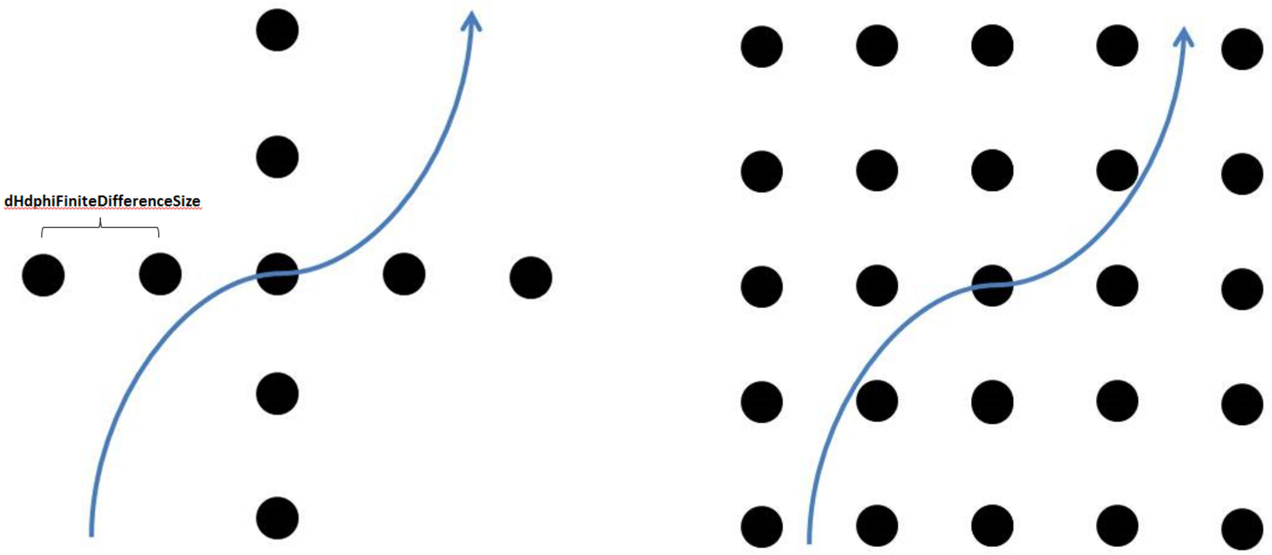

Figure 4: An illustration of the finite difference mesh in 2-dimensional flux space for evaluating the partial derivatives in $C_t(n, \vec{r})$. $H_0(\phi(t))$ is diagonalized at each one of these points. This mesh must be rebuilt at each stop along the ideal flux path. Here, **dHdphiSamplingPoints**$= 5$. Left: **taylorExpansionOrder** $= 1$. Right: **taylorExpansionOrder** ≥ 2

3.2 dHdphiSamplingPoints and dHdphiFiniteDifferenceSize

To evaluate the coefficients of the taylor series expansion of a Hamiltonian in the instantaneous eigenbasis, $H(\phi(t))$, we must differentiate in flux space around the ideal flux path:

$$C_t(n, \vec{r}) = \frac{\partial^n H(\vec{\phi}(t) + \vec{\xi}(t))}{\partial\xi_{r_1}\partial\xi_{r_2}...\partial\xi_{r_n}}|_{\vec{\xi}=\vec{0}} \tag{21}$$

If the user chooses **taylorExpansionOrder** $= 1$, the program will build $H(\vec{\phi}(t) + \vec{\xi}(t))$ at discrete intervals along each direction in flux space, and apply a central finite difference formula. If the user chooses **taylorExpansionOrder** ≥ 2, the $C_t(n, \vec{r})$ matrices will contain mixed partial derivatives and the program will build $H(\vec{\phi}(t) + \vec{\xi}(t))$ along a hyper-dimensional mesh. See Fig. (3.2).

The user must choose an odd number for the grid size, **dHdphiSamplingPoints** and a Real number for the spacing between grid points **dHdphiFiniteDifferenceSize**.

The user should pick these parameters such that error is minimized in the numerical derivatives, i.e. pick a point spacing that accurately captures the changes in the Hamiltonian, and then choose a number of sampling points that suits that point spacing. Note that increasing **dHdphiSamplingPoints** will increase the computational workload of the program.

4 Some Final Notes

4.1 Communication between successive diagonalization of $H_0(\phi(t))$

A particularly tedious challenge in writing this code was building efficient communication between successive diagonalizations of $H_0(\phi(t))$. Crossing-eigenvalue degeneracies along a flux path are predicted with extrapolation, and then corrected. In addition, crossing-eigenvalue degeneracies are checked along the *initial* interval used to calculate the partial derivatives in the coefficient matrices, $C_t(n, \vec{r})$. The correct ordering of the lowest eigenvalues is maintained over the time interval.

Discrete jumps in the global phases of the eigenvectors are also corrected. I.e, for any matrix A and eigenvalue λ,

$$\begin{aligned} \text{If } A\left|\psi\right\rangle &= \lambda\left|\psi\right\rangle \\ \text{then } A(e^{i\phi}\left|\psi\right\rangle) &= \lambda(e^{i\phi}\left|\psi\right\rangle) \end{aligned} \tag{22}$$

for any real ϕ. Large discontinuities in the global phases between successive sets of eigenvectors will result in an incorrect derivative in the non-adiabatic operator, $\frac{dR^{\dagger}(\vec{\phi}(t))}{dt}$ in the off-diagonal entries. Large discontinuities are avoided by adjusting the global phase of each eigenvector such that the first entry of each eigenvector is a positive, real number. For a complex Hamiltonian, this will cause an overall "drift" in the phases of the eigenbasis, and therefore in the phases of the off-diagonal elements of the expansion coefficients. This something to keep in mind if you want to compare the output of this code to an analytic example (see the notebook *Coefficient Test.nb* for such a comparison). We can convert between the initial ($\left|\psi(t)\right\rangle$) and instantaneous ($\left|\psi(t)\right\rangle'$) eigenbasis via the transformation

$$\left|\psi(t)\right\rangle' = R(\phi(t))\left|\psi(t)\right\rangle \qquad \left|\psi(t)\right\rangle = R^{\dagger}(\phi(t))\left|\psi(t)\right\rangle' \tag{23}$$

4.2 Interfacing with *Mathematica*

As it stands, the code is about as fast as it's going to be without significant overhaul. Fortunately, WSTP is a well-suited to inter-program communication, and all data exchanged between the C++ binary and *Mathematica* notebook is highly efficient. Because Eigen3 has no Arnoldi partial diagonalization routine, diagonalizing $H_0(\phi(t))$ in *Mathematica* is the best way to do this right now anyway. I tried porting more of my code to lessen the communication load for WSTP, and possibly work exclusively in *Mathematica*, but this actually lead to significant slowdown. In particular, large matrix addition in Eigen3 is fast enough to warrant keeping most of my program as a C++ binary.

Further, I think usability is good as the program stands now. Dr. Kerman is able to write his Hamiltonians in *Mathematica* as he prefers to do, and at least enjoy some scalability allowed by the low-level control of C++.

4.3 Acknowledgments

I want to thank Group 89 for giving me the opportunity to work at Lincoln Lab this summer, and I especially want to thank Dr. Obenland and Dr. Kerman for putting up with an intern; I

know you guys are busy working on other things. I hope this code is useful to you guys, if you want me to run any simulations, or perform any maintenance on this, or add any additional features, please do not hesitate to contact me.

Thanks again,
–Jake

References

[1] Fornberg, Bengt (1988), "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids", Mathematics of Computation 51 (184): 699706