Appendix A: Computational Details and Explicit Equations Used

In this section, we first present the governing equations for the flow in our thermal convection loop experiment. A spatial and temporal discretization of the governing equations is then necessary so that they may be solved numerically. After discretization, we must specify the boundary conditions. With the mesh and boundary conditions in place, we can then simulate the flow with a computational fluid dynamics solver.

We now discuss the equations, mesh, boundary conditions, and solver in more detail. With these considerations, we present our simulations of the thermosyphon. For a complete derivation of the equations used, see [32].

1. Governing Equations

We consider the incompressible Navier-Stokes equations with the Boussinesq approximation to model the flow of water inside a thermal convection loop. Here we present the main equations that are solved numerically, noting the assumptions that were necessary in their derivation. In Appendix C a more complete derivation of the equations governing the fluid flow, and the numerical for our problem, is presented. In standard notation, for u, v, w the velocity in the x, y, z direction, respectively, the continuity equation for an incompressible fluid is

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0. \tag{A1}$$

The momentum equations, presented compactly in tensor notation with bars representing averaged quantities, are given by

$$\rho_{\text{ref}} \left(\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial}{\partial x_j} \left(\bar{u}_j \bar{u}_i \right) \right) = -\frac{\partial \bar{p}}{\partial x_i} + \mu \frac{\partial \bar{u}_i}{\partial x_j^2} + \bar{\rho} g_i \tag{A2}$$

for ρ_{ref} the reference density, ρ the density from the Boussinesq approximation, p the pressure, μ the viscocity, and g_i gravity in the i-direction. Note that $g_i = 0$ for $i \in \{x, y\}$ since gravity is assumed to be the z direction. The energy equation is given by

$$\frac{\partial}{\partial t} \left(\rho \bar{e} \right) + \frac{\partial}{\partial x_j} \left(\rho \overline{e} \overline{u}_j \right) = -\frac{\partial q_k^*}{\partial x_k} - \frac{\partial \overline{q}_k}{\partial x_k} \tag{A3}$$

for e the total internal energy and q the flux (where $q = \overline{q} + q^*$ is the averaging notation).

2. Implementation

The PISO (Pressure-Implicit with Splitting of Operators) algorithm derives from the work of [20], and is complementary to the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) [30] iterative method. The main difference of the PISO and SIMPLE algorithms is that in the PISO, no under-relaxation is applied and the momentum corrector step is performed more than once [12]. They sum up the algorithm in nine steps:

- Set the boundary conditions
- Solve the discretized momentum equation to compute an intermediate velocity field
- Compute the mass fluxes at the cell faces
- Solve the pressure equation
- Correct the mass fluxes at the cell faces
- Correct the velocity with respect to the new pressure field
- Update the boundary conditions
- Repeat from step #3 for the prescribed number of times
- Repeat (with increased time step).

The solver itself has 647 dependencies, of which I present only a fraction. The main code is straight forward, relying on include statements to load the libraries and equations to be solved.

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "RASModel.H" // AJR edited 2013-10-14
#include "radiationModel.H"
#include "fvIOoptionList.H"
#include "pimpleControl.H"
```

The main function is then

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "readGravitationalAcceleration.H"
    #include "createFields.H"
    #include "createIncompressibleRadiationModel.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"
    #include "readTimeControls.H"
    #include "CourantNo.H"
    #include "setInitialDeltaT.H"
    pimpleControl pimple(mesh);
```

We then enter the main loop. This is computed for each time step, prescribed before the solver is applied. Note that the capacity is available for adaptive time steps, choosing to keep the Courant number below some threshold, but I do not use this. For the distributed ensemble of model runs, it is important that each model complete in nearly the same time, so that the analysis is not waiting on one model and therefore under-utilizing the available resources.

```
    while (runTime.loop())
    {
        #include "readTimeControls.H"
        #include "CourantNo.H"
        #include "setDeltaT.H"
        while (pimple.loop())
        {
            #include "UEqn.H"
            #include "TEqn.H"
            while (pimple.correct())
            {
                #include "pEqn.H"
            }
        }
        if (pimple.turbCorr())
        {
            turbulence->correct();
        }
```

Opening up the equation for U we see that Equation

```
    // Solve the momentum equation
    fvVectorMatrix UEqn
    (
        fvm::ddt(U)
      + fvm::div(phi, U)
      + turbulence->divDevReff(U)
     ==
        fvOptions(U)
    );
    UEqn.relax();
    fvOptions.constrain(UEqn);
    if (pimple.momentumPredictor())
    {
        solve
        (
            UEqn
         ==
            fvc::reconstruct
            (
                (
                  - ghf*fvc::snGrad(rhok)
                  - fvc::snGrad(p_rgh)
                )*mesh.magSf()
            )
        );
```

```
        fvOptions.correct(U);
    }
```

Solving for T is

```
{
    alphat = turbulence->nut()/Prt;
    alphat.correctBoundaryConditions();
    volScalarField alphaEff("alphaEff", turbulence->nu()/Pr + alphat);
    fvScalarMatrix TEqn
    (
        fvm::ddt(T)
      + fvm::div(phi, T)
      - fvm::laplacian(alphaEff, T)
     ==
        radiation->ST(rhoCpRef, T)
      + fvOptions(T)
    );
    TEqn.relax();
    fvOptions.constrain(TEqn);
    TEqn.solve();
    radiation->correct();
    fvOptions.correct(T);
    rhok = 1.0 - beta*(T - TRef); // Boussinesq approximation
}
```

Finally, we solve for the pressure p in "pEqn.H":

```
{
    volScalarField rAU("rAU", 1.0/UEqn.A());
    surfaceScalarField rAUf("Dp", fvc::interpolate(rAU));
    volVectorField HbyA("HbyA", U);
    HbyA = rAU*UEqn.H();
    surfaceScalarField phig(-rAUf*ghf*fvc::snGrad(rhok)*mesh.magSf());
    surfaceScalarField phiHbyA
    (
        "phiHbyA",
        (fvc::interpolate(HbyA) & mesh.Sf())
      + fvc::ddtPhiCorr(rAU, U, phi)
      + phig
    );
    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix p_rghEqn
        (
            fvm::laplacian(rAUf, p_rgh) == fvc::div(phiHbyA)
        );
        p_rghEqn.setReference(pRefCell, getRefCellValue(p_rgh, pRefCell));
        p_rghEqn.solve(mesh.solver(p_rgh.select(pimple.finalInnerIter())));
        if (pimple.finalNonOrthogonalIter())
        {
            // Calculate the conservative fluxes
            phi = phiHbyA - p_rghEqn.flux();
            // Explicitly relax pressure for momentum corrector
            p_rgh.relax();
            // Correct the momentum source with the pressure gradient flux
            // calculated from the relaxed pressure
            U = HbyA + rAU*fvc::reconstruct((phig - p_rghEqn.flux())/rAUf);
            U.correctBoundaryConditions();
        }
    }
    #include "continuityErrs.H"
    p = p_rgh + rhok*gh;
    if (p_rgh.needReference())
    {
        p += dimensionedScalar
        (
            "p",
            p.dimensions(),
            pRefValue - getRefCellValue(p, pRefCell)
        );
        p_rgh = p - rhok*gh;
    }
}
```

The final operation being the conversion of pressure to hydrostatic pressure,

$$p_{\mathrm{rgh}} = p - \rho_k g_h.$$

This "pEqn.H" is then re-run until convergence is achieved, and the PISO loop begins again.

Appendix B: The Ehrhard and Müller Equations

Following the derivation by Harris [17], itself a representation of the derivation of Gorman [16] and namesakes Ehrhard and Müller [10], we derive the equations governing a closed loop thermosyphon.

Similar to the derivation of the governing equations of computational fluid dynamics, we start with a small but finite volume inside the loop. Here, however, the volume is described by $\pi r^2 R\mathrm{d}\phi$ for r the interior loop size (such that πr^2 is the area of a slice) and $R\mathrm{d}\phi$ the arc length (width) of the slice. Newton's second law states that momentum is conserved, such that the sum of the forces acting upon our finite volume is equal to the change in momentum of this volume. Therefore we have the basic starting point for forces $\sum F$ and velocity u as

$$\sum F = \rho\pi r^2 R\mathrm{d}\phi\frac{\mathrm{d}u}{\mathrm{d}t}. \tag{B1}$$

The sum of the forces is $\sum F = F_{\{p,f,g\}}$ for net pressure, fluid shear, and gravity, respectively. We write these as

$$F_p = -\pi r^2\mathrm{d}\phi\frac{\partial p}{\partial\phi} \tag{B2}$$

$$F_w = -\rho\pi r^2\mathrm{d}\phi f_w \tag{B3}$$

$$F_g = -\rho\pi r^2\mathrm{d}\phi g\sin(\phi) \tag{B4}$$

where $\partial p/\partial\phi$ is the pressure gradient, f_w is the wall friction force, and $g\sin(\phi)$ is the vertical component of gravity acting on the volume.

We now introduce the Boussinesq approximation which states that both variations in fluid density are linear in temperature T and density variation is insignificant except when multiplied by gravity. The consideration manifests as

$$\rho = \rho(T) \simeq \rho_{\mathrm{ref}}(1 - \beta(T - T_{\mathrm{ref}})$$

where ρ_0 is the reference density and T_{ref} is the reference temperature, and β is the thermal expansion coefficient. The second consideration of the Boussinesq approximation allows us to replace ρ with this ρ_{ref} in all terms except for F_g. We now write momentum equation as

$$-\pi r^2\mathrm{d}\phi\frac{\partial p}{\partial\phi} - \rho_{\mathrm{ref}}\phi r^2 R\mathrm{d}\phi f_w - \rho_{\mathrm{ref}}(1 - \rho(T - T_{\mathrm{ref}}))\pi r^2 R\mathrm{d}\phi g\sin(\phi) = \rho_{\mathrm{ref}}\pi r^2 R\mathrm{d}\phi\frac{\mathrm{d}u}{\mathrm{d}t}. \tag{B5}$$

Canceling the common πr^2, dividing by R, and pulling out $\mathrm{d}\phi$ on the LHS we have

$$-\mathrm{d}\phi\left(\frac{\partial p}{\partial\phi}\frac{1}{R} - \rho_{\mathrm{ref}}f_w - \rho_{\mathrm{ref}}(1 - \rho(T - T_{\mathrm{ref}}))g\sin(\phi)\right) = \rho_{\mathrm{ref}}\mathrm{d}\phi\frac{\mathrm{d}u}{\mathrm{d}t}. \tag{B6}$$

We integrate this equation over ϕ to eliminate many of the terms, specifically we have

$$\int_0^{2\pi} -\mathrm{d}\phi\frac{\partial p}{\partial\phi}\frac{1}{R} \to 0$$

$$\int_0^{2\pi} -\mathrm{d}\phi\rho_{\mathrm{ref}}g\sin(\phi) \to 0$$

$$\int_0^{2\pi} -\mathrm{d}\phi\rho_{\mathrm{ref}}\beta T_{\mathrm{ref}}g\sin(\phi) \to 0.$$

Since u (and hence $\frac{\mathrm{d}u}{\mathrm{d}\phi}$) and f_w do not depend on ϕ, we can pull these outside an integral over ϕ and therefore the momentum equation is now

$$2\pi f_w\rho_0 + \int_0^{2\pi}\mathrm{d}\phi\rho_{\mathrm{ref}}\beta Tg\sin(\phi) = 2\pi\frac{\mathrm{d}u}{\mathrm{d}\phi}\rho_{\mathrm{ref}}.$$

Diving out 2π and pull constants out of the integral we have our final form of the momentum equation

$$f_w \rho_{\text{ref}} + \frac{\rho_{\text{ref}} \beta g}{2\pi} \int_0^{2\pi} \mathrm{d}\phi T \sin(\phi) = \frac{\mathrm{d}u}{\mathrm{d}\phi} \rho_{\text{ref}}. \tag{B7}$$

Now considering the conservation of energy within the thermosyphon, the energy change within a finite volume must be balanced by transfer within the thermosyphon and to the walls. The internal energy change is given by

$$\rho_{\text{ref}} \pi r^2 R \mathrm{d}\phi \left(\frac{\partial T}{\partial t} + \frac{u}{R} \frac{\partial T}{\partial \phi} \right) \tag{B8}$$

which must equal the energy transfer through the wall, which is, for T_w the wall temperature:

$$\dot{q} = -\pi r^2 R \mathrm{d}\phi h_w (T - T_w). \tag{B9}$$

Combining Equations B8 and B9 (and canceling terms) we have the energy equation:

$$\left(\frac{\partial T}{\partial t} + \frac{u}{R} \frac{\partial T}{\partial \phi} \right) = \frac{-h_w}{\rho_{\text{ref}} c_p} \left(T - T_w \right). \tag{B10}$$

The f_w which we have yet to define and h_w are fluid-wall coefficients and can be described by [10]:

$$h_w = h_{w_0} \left(1 + K h(|x_1|) \right)$$
$$f_w = \frac{1}{2} \rho_{\text{ref}} f_{w_0} u.$$

We have introduced an additional function h to describe the behavior of the dimensionless velocity $x_1 \alpha u$. This function is defined piece-wise as

$$h(x) = \begin{cases} x^{1/3} & \text{when } x \geq 1 \\ p(x) & \text{when } x < 1 \end{cases}$$

where $p(x)$ can be defined as $p(x) = \left(44x^2 - 55x^3 + 20x^4 \right)/9$ such that p is analytic at 0 [17].

Taking the lowest modes of a Fourier expansion for T for an approximate solution, we consider:

$$T(\phi, t) = C_0(t) + S(t) \sin(\phi) + C(t) \cos(\phi). \tag{B11}$$

By substituting this form into Equations B7 and B10 and integrating, we obtain a system of three equations for our solution. We then follow the particular nondimensionalization choice of Harris et al such that we obtain the following ODE system, which we refer to as the Ehrhard-Müller equations:

$$\frac{\mathrm{d}x_1}{\mathrm{d}t'} = \alpha(x_2 - x_1), \tag{B12}$$
$$\frac{\mathrm{d}x_2}{\mathrm{d}t'} = \beta x_1 - x_2(1 + K h(|x_1|)) - x_1 x_3, \tag{B13}$$
$$\frac{\mathrm{d}x_3}{\mathrm{d}t'} = x_1 x_2 - x_3(1 + K h(|x_1|)). \tag{B14}$$

The nondimensionalization is given by the change of variables

$$t' = \frac{h_{w_0}}{\rho_{\text{ref}} c_p} t, \tag{B15}$$
$$x_1 = \frac{\rho_{\text{ref}} c_p}{R h_{w_0}} u, \tag{B16}$$
$$x_2 = \frac{1}{2} \frac{\rho_{\text{ref}} c_p \beta g}{R h_{w_0} f_{w_0}} \Delta T_{3-9}, \tag{B17}$$
$$x_3 = \frac{1}{2} \frac{\rho_{\text{ref}} c_p \beta g}{R h_{w_0} f_{w_0}} \left(\frac{4}{\pi} \Delta T_w - \Delta T_{6-12} \right) \tag{B18}$$

and

$$\alpha = \frac{1}{2} R c_p f_{w_0} / h_{w_0}, \tag{B19}$$

$$\gamma = \frac{2}{\pi} \frac{\rho_{\text{ref}} c_p \beta g}{R h_{w_0} f_{w_0}} \Delta T_w. \tag{B20}$$

Through careful consideration of these non-dimensional variable transformations we verify that x_1 is representative of the mean fluid velocity, x_2 of the temperature difference between the 3 and 9 o'clock positions on the thermosyphon, and x_3 the deviation from the vertical temperature profile in a conduction state [17].

Appendix C: Data Assimiliation

The TLM is the model which advances an initial perturbation $\delta\mathbf{x}_i$ at timestep i to a final perturbation $\delta\mathbf{x}_{i+1}$ at timestep $i+1$. The dynamical system we are interested in, Lorenz '63, is given as a system of ODE's:

$$\frac{d\mathbf{x}}{dt} = F(\mathbf{x}).$$

We integrate this system using a numerical scheme of our choice (in the given examples we use a second-order Runge-Kutta method), to obtain a model M discretized in time.

$$\mathbf{x}(t) = M[\mathbf{x}(t_0)].$$

Introducing a small perturbation $\mathbf{y}$, we can approximate our model M applied to $\mathbf{x}(t_0) + \mathbf{y}(t_0)$ with a Taylor series around $\mathbf{x}(t_0)$:

$$\begin{aligned} M[\mathbf{x}(t_0) + \mathbf{y}(t_0)] &= M[\mathbf{x}(t_0)] + \frac{\partial M}{\partial \mathbf{x}}\mathbf{y}(t_0) + O[\mathbf{y}(t_0)^2] \\ &\approx \mathbf{x}(t) + \frac{\partial M}{\partial \mathbf{x}}\mathbf{y}(t_0). \end{aligned}$$

We can then solve for the linear evolution of the small perturbation $\mathbf{y}(t_0)$ as

$$\frac{d\mathbf{y}}{dt} = \mathbf{J}\mathbf{y} \tag{C1}$$

where $\mathbf{J} = \partial F/\partial \mathbf{x}$ is the Jacobian of F. We can solve the above system of linear ordinary differential equations using the same numerical scheme as we did for the nonlinear model.

One problem with solving the system of equations given by Equation C1 is that the Jacobian matrix of discretized code is not necessarily identical to the discretization of the Jacobian operator for the analytic system. This is a problem because we need to have the TLM of our model M, which is the time-space discretization of the solution to $d\mathbf{x}/dt = F(\mathbf{x})$. We can apply our numerical method to the $d\mathbf{x}/dt = F(\mathbf{x})$ to obtain M explicitly, and then take the Jacobian of the result. This method is, however, prohibitively costly, since Runge-Kutta methods are implicit. It is therefore desirable to take the derivative of the numerical scheme directly, and apply this differentiated numerical scheme to the system of equations $F(\mathbf{x})$ to obtain the TLM. A schematic of this scenario is illustrated in Figure S2. To that the derivative of numerical code for implementing the EKF on models larger than 3 dimensions (i.e. global weather models written in Fortan), automatic code differentiation is used [31].

To verify our implementation of the TLM, we propagate a small error in the Lorenz '63 system and plot the difference between that error and the TLM predicted error, for each variable (Figure S1).

With a finite ensemble size, the ensemble method is only an approximation and therefore in practice it often fails to capture the full spread of error. To better capture the model variance, additive and multiplicative inflation factors are used to obtain a good estimate of the error covariance matrix (Section 2.6). The spread of ensemble members in the x_1 variable of the Lorenz model, as distance from the analysis, can be seen in Figure S3.

In computing the error covariance $\mathbf{P}_f$ from the ensemble, we wish to add up the error covariance of each forecast with respect to the mean forecast. But this would underestimate the error covariance, since the forecast we're comparing against was used in the ensemble average (to obtain the mean forecast). Therefore, to compute the error covariance matrix for each forecast, that forecast itself is excluded from the ensemble average forecast.

We can see the classic spaghetti of the ensemble with this filter implemented on Lorenz 63 in Figure S4.

We denote the forecast within an ensemble filter as the average of the individual ensemble forecasts, and an explanation for this choice is substantiated by Burgers [5]. The general EnKF which we use is most similar to that of Burgers. Many algorithms based on the EnKF have been proposed and include the Ensemble Transform Kalman Filter (ETKF) [29], Ensemble Analysis Filter (EAF) [1], Ensemble Square Root Filter (EnSRF) [36], Local Ensemble Kalman Filter (LEKF) [29], and the Local Ensemble Transform Kalman Filter (LETKF) [19]. A comprehensive overview through 2003 is provided by Evensen [11]. For further details on the most advanced methods, beyond what is provided in the body of the paper, we direct the reader the above references and the derivations provided in [32].

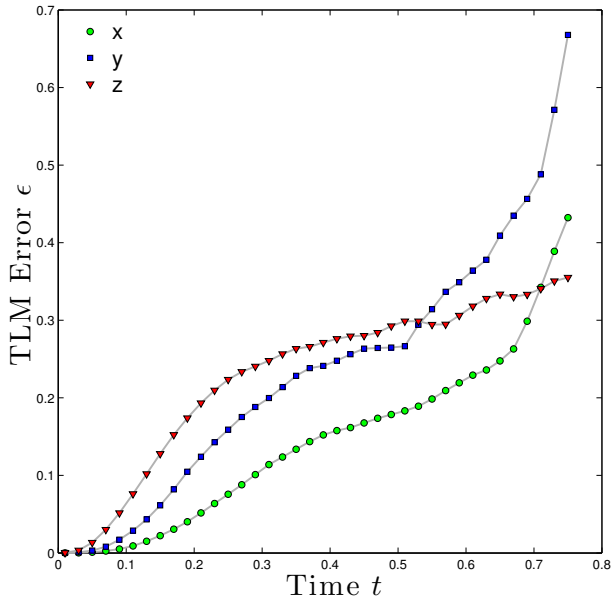

FIG. S1: The future error predicted by the TLM is compared to the error growth in Lorenz '63 system for an initial perturbation with standard deviation of 0.1, averaged over 1000 TLM integrations. The ϵ is not the error predicted by the TLM, but rather the error of the TLM in predicting the error growth.

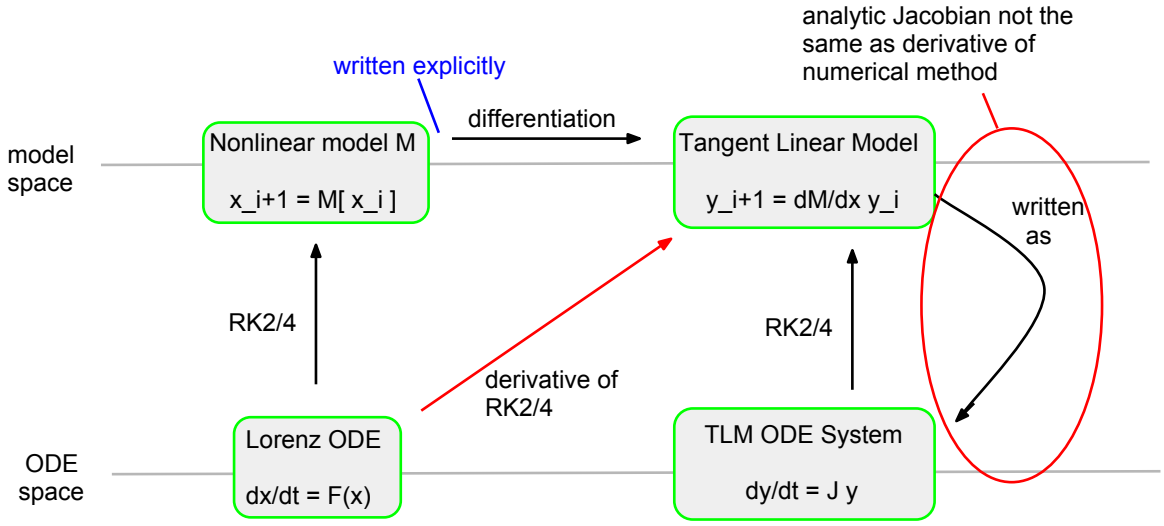

FIG. S2: An explanation of how and why the best way to obtain a TLM is with a differentiated numerical scheme.

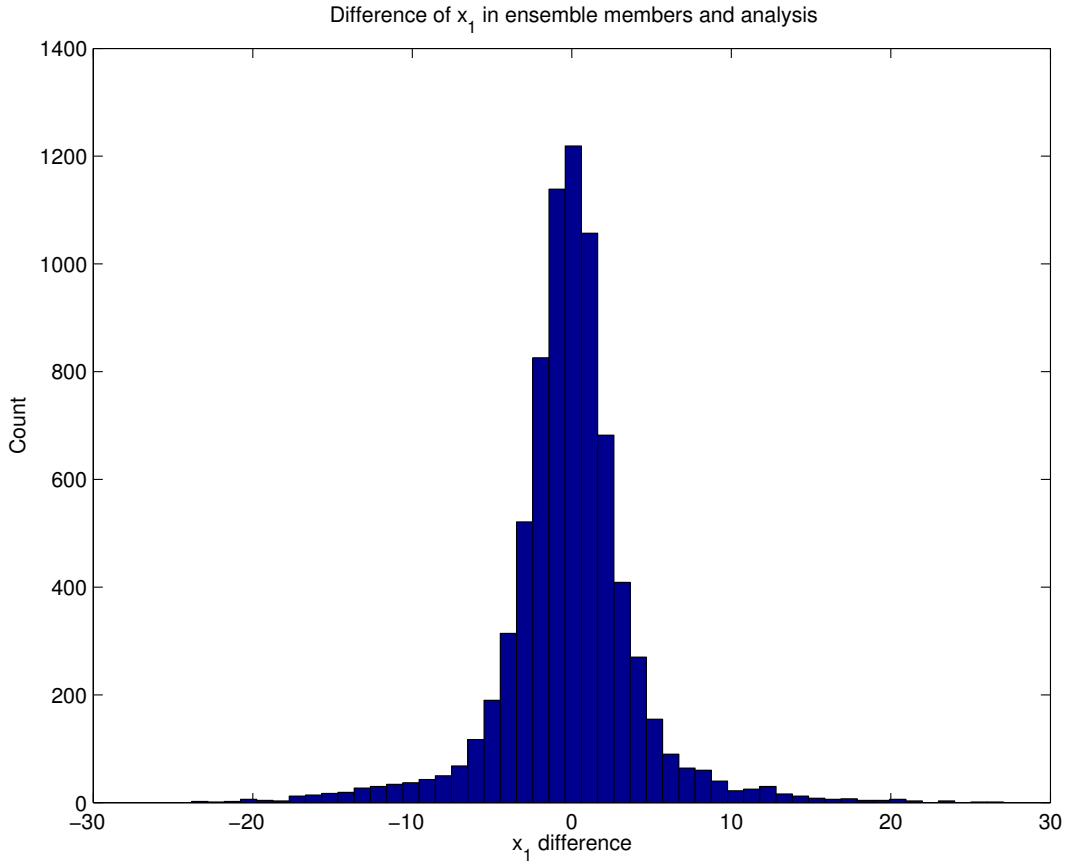

FIG. S3: The difference of ensemble forecasts from the analysis is reported for 760 assimilation windows in one model run of length 200, with 10 ensemble members and an assimilation window of length 0.261. This has the same shape of as the difference between ensemble forecasts and the mean of the forecasts (not shown). This spread of ensemble forecasts is what allows us to estimate the error covariance of the forecast model, and appears to be normally distributed.

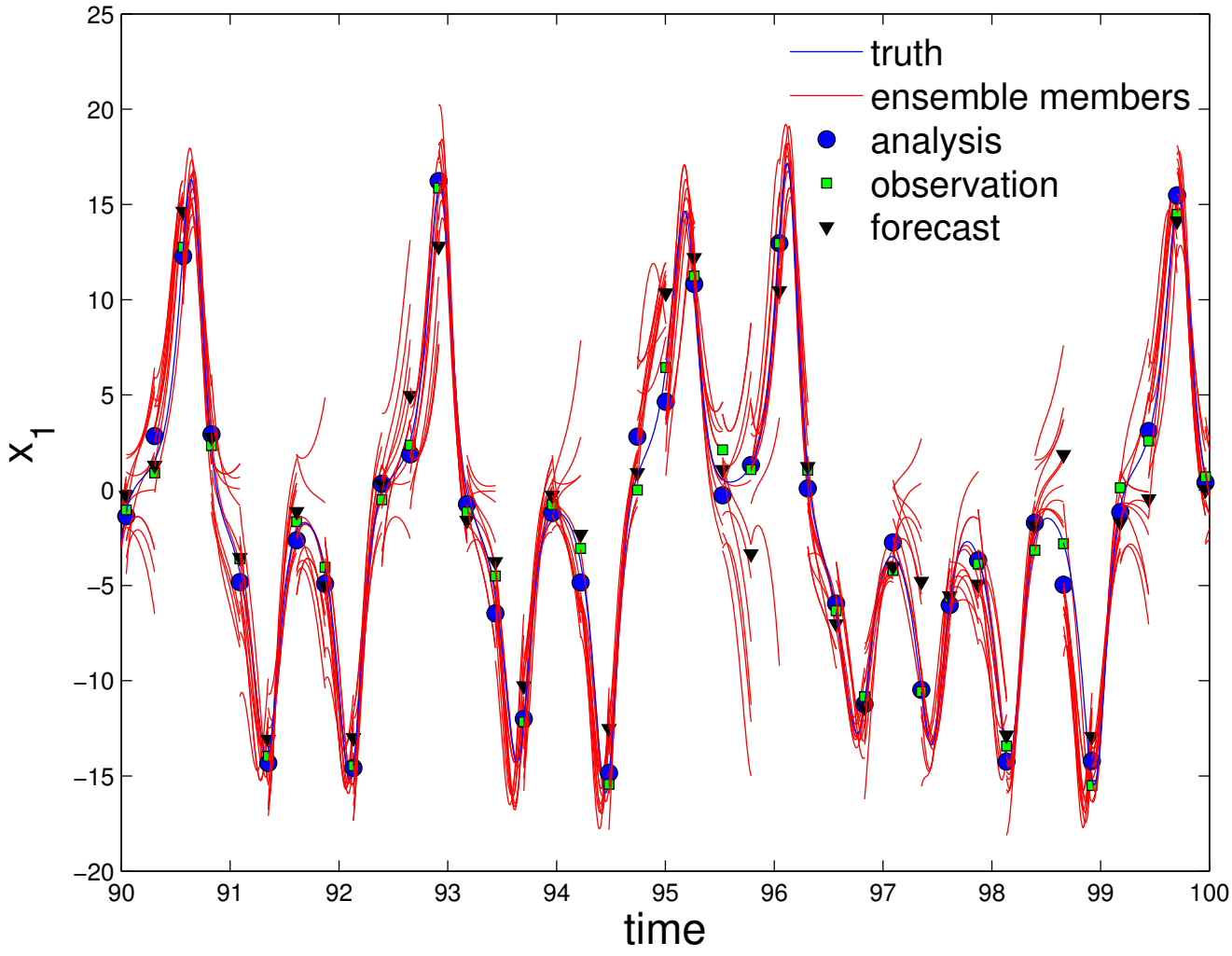

FIG. S4: A sample time-series of the ensembles used in the EnKF. In all tests, as seen here, 10 ensemble members are used. For this run, 384 assimilation cycles were performed with a window length of 0.26 model time units.

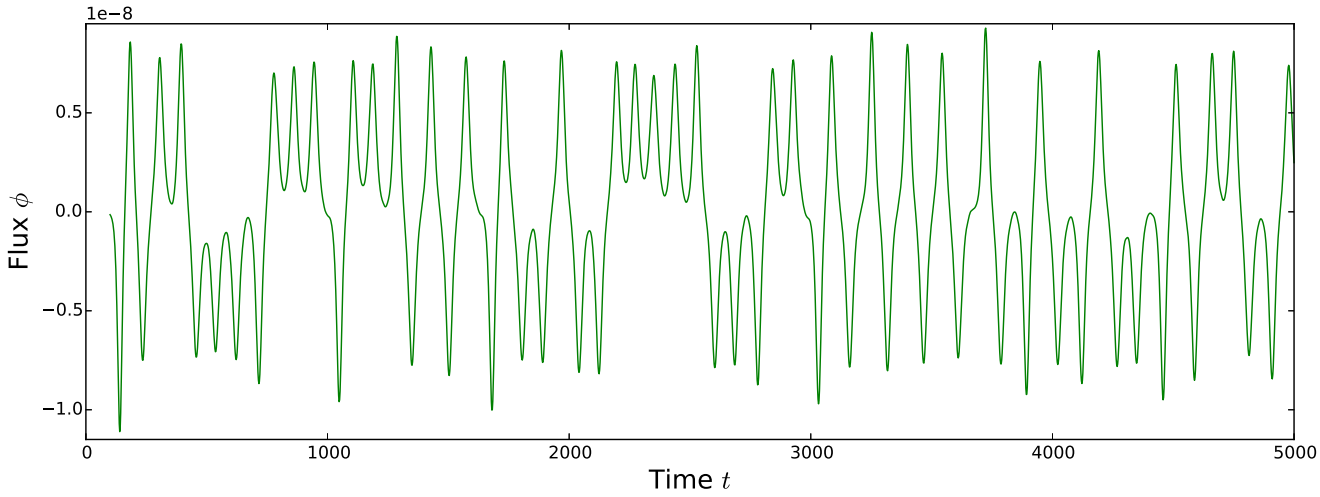

FIG. S5: Flux timeseries on which DMD was performed.

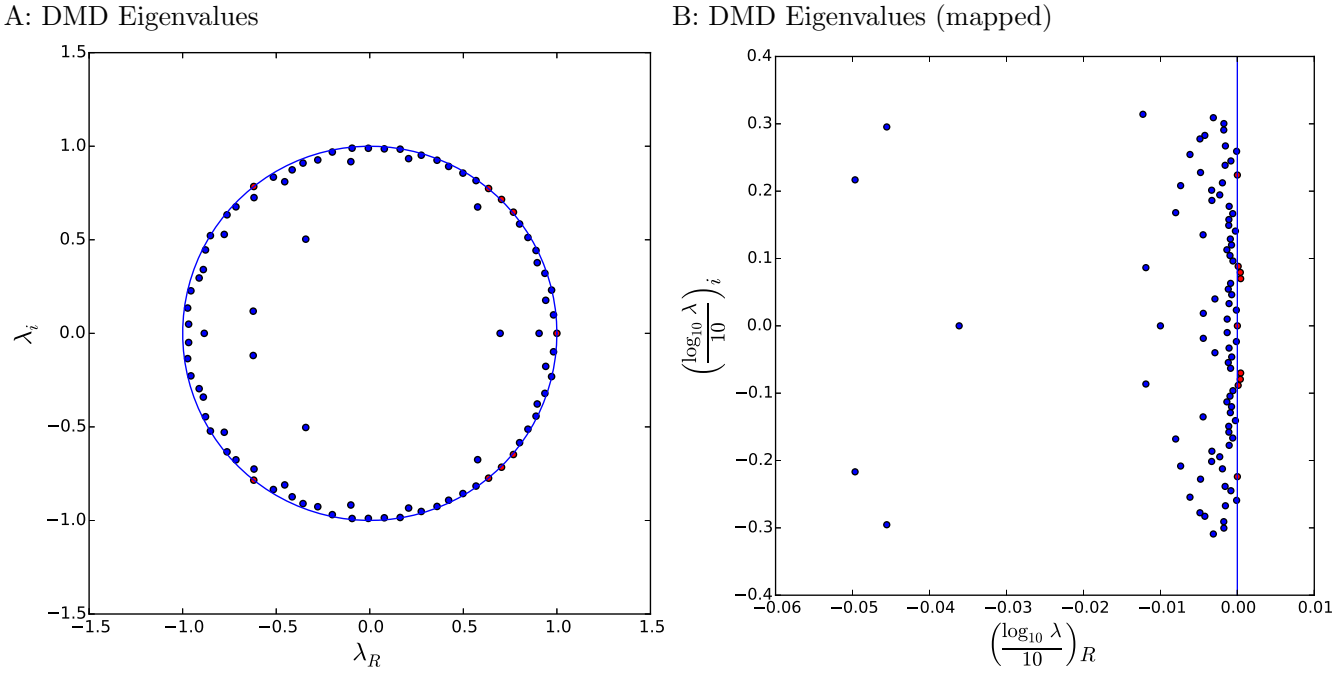

FIG. S6: Eigenvalues of DMD Modes.

Appendix D: Additional DMD Details

The general algoritm for DMD was provided in Section III(C), and here we supply more results of the DMD procedure. The timeseries from which we computed the decomposition is shownn in Figure S5.

The real and imaginary components of the DMD eigenvalues are shown in both unmapped and mapped form in Figure S6.