



Extending THCM with an atmospheric vapor model

Sander Land



Extending THCM with an atmospheric vapor model

Sander Land

Supervisors:

F.W. Wubs and H. Bekker

University of Groningen

Institute for Mathematics and Computing Science

P.O. Box 407

9700 AK Groningen

The Netherlands

August 2007

Contents

1	Introduction	3
2	Modelling	5
2.1	Transport	6
2.2	Evaporation	6
2.3	Precipitation	7
2.4	Scaling and summary of equations	8
2.5	Influence on salinity	8
3	Discretization and Implementation	9
3.1	Requirements	9
3.2	Understanding THCM	9
3.2.1	The array of local coefficients A1	10
3.2.2	Compressed sparse row format	10
3.2.3	Grid	11
3.3	Implementation	11
3.3.1	Tools	11
3.3.2	Diffusion	12
3.3.3	Evaporation	13
3.3.4	Salinity	13
3.3.5	Precipitation	13
4	Visualization	19
4.1	Tools used	19
4.2	Implementation	19
4.2.1	Reading data	19
4.2.2	Graphical User Interface	20
4.2.3	Plotting	21
5	Results	23
5.1	Setup	23
5.2	Timings	24
5.3	Comparison	25
5.3.1	Comparison of v2 and v3 precipitation models	26
5.3.2	Influence on oceanic flow	28
5.3.3	Effect of the wind velocity on the humidity distribution	33
5.4	Discussion	33

A Fortran code	37
A.1 v2	37
A.2 v3	43
B Ruby Code	45
B.1 Visualization Tool	45

Chapter 1

Introduction

THCM [1] (Thermohaline circulation model) is an oceanographic model used to study the oceanic flow and its influence on climate changes. The model currently implements oceanic flow, with the velocity vector, pressure, salinity and temperature. It also models atmospheric temperature.

To increase the accuracy and reduce the dependency on external data like heat and salinity forcing, some extensions are planned: a sea-ice model and models for water vapor and CO₂ distributions in the atmosphere.

This thesis describes one of these extensions, the addition of a model for the water vapor distribution and its effect on oceanic flow.

In chapter two, a mathematical model of the evaporation, precipitation and transport of water vapor in the atmosphere will be presented. In the next chapter the discretization of this model will be discussed, as well as some challenges associated with implementing the model in THCM.

In chapter four, a simple visualization tool that was written to study the results will be presented. In the last chapter this tool will be used to show the results.

Chapter 2

Modelling

The model used is the one proposed by [2]. It already takes some of the limitations and requirements for THCM into account. The model will be reproduced here for completeness, focussing on the parts that are necessary for the implementation.

One of the main limitations of THCM with regards to atmospheric modelling is that there is only a single atmospheric layer. Because of this, the quantity modelled is not the humidity, but the *vertically averaged* humidity in the atmosphere (in kg/kg), denoted by q .

The average amount of vapor in a column of air above a specific point on the ocean's surface can change because of transport from neighbouring points, as well as from evaporation and precipitation over the ocean.

The equation modelling this change of humidity over time is

$$H_q \rho_a \frac{\partial q}{\partial t} = M + \rho_0(E - P)$$

where

- H_q is the height scale, i.e. the height of the part atmosphere that is considered to have any vapor at all (1800m),
- ρ_a is the atmospheric density (1.25 kg/m³), ρ_0 is the oceanic density (1000 kg/m³),
- M is a term modelling the transport of vapor in the atmosphere (in kg/m²s),
- and E and P are the evaporation and precipitation rates (in m/s).

This formula shows the change in total humidity ($H_q \rho_a \frac{\partial q}{\partial t}$) in the 1800m high column of atmosphere to be the total amount of transport in this entire column (M) plus the difference between the evaporation and precipitation.

Because THCM needs the Jacobian of the time derivative, the equation has to be linearized. This is done by taking $q = q_0 + q'$, $E = E_0 + E'$, $P = P_0 + P'$ and results in:

$$H_q \rho_a \frac{\partial q'}{\partial t} = M' + \rho_0(E' - P')$$

2.1 Transport

The transport of water vapor in the atmosphere is modelled in a very simple way, using a diffusion equation:

$$M' = H_q \rho_a \left(\kappa_x \frac{\partial^2 q'}{\partial x^2} + \kappa_y \frac{\partial^2 q'}{\partial y^2} \right)$$

where κ_x, κ_y are the zonal and meridional eddy-diffusivities for vapor. The factor $H_q \rho_a$ shows that this is the transport for all the vapor in the entire column of air.

This equation has some physical counterpart in the form of diffusion due to turbulent eddies, but is a large oversimplification as wind forcing is completely ignored.

The boundary condition used here is $\frac{\partial q}{\partial n} = 0$, i.e. there is no loss of water vapor at the boundaries.

2.2 Evaporation

The evaporation E represents the speed in m/s at which the ocean 'flows into' the atmosphere. The standard equation for this is

$$E = \frac{\rho_a}{\rho_0} C_E U_a(x) (q_{\text{sat}}(T) - q)$$

where

- T is the oceanic surface temperature,
- $U_a(x)$ is the scalar wind velocity (in m/s),
- q_{sat} is the saturation specific humidity (in kg/kg),
- and C_E is the Dalton number.

Linearising this equation gives:

$$E' = \left. \frac{\partial E}{\partial T} \right|_{T_0, q_0} T' + \left. \frac{\partial E}{\partial q} \right|_{T_0, q_0} q' = \varepsilon_T T' + \varepsilon_q q'$$

ε_q does not depend on T_0, q_0 :

$$\varepsilon_q = -C_E U_a(x) \frac{\rho_a}{\rho_0}$$

But ε_T does:

$$\varepsilon_T = C_E U_a(x) \frac{\rho_a}{\rho_0} \frac{dq_{\text{sat}}}{dT}(T_0)$$

So $\frac{dq_{\text{sat}}}{dT}(T_0)$ needs to be determined. This is possible by using the relation between the saturation specific humidity q_{sat} and the saturation vapor pressure e_{sat} :

$$q_{\text{sat}} = \frac{e_{\text{sat}}}{p}$$

where p is the atmospheric pressure (1000 hPa) and ϵ is the mixing ratio of water vapor and dry air (0.622).

There are two alternatives for determining e_{sat} . Both of these involve the saturation vapor pressure at the triple point ($T_{\text{tr}} = 273.16\text{K}$, $e_{\text{sat}}(T_{\text{tr}}) = 6.11\text{hPa}$):

$$\frac{de_{\text{sat}}}{dT} = e_{\text{sat, tr}} f(T_0)$$

The two alternatives for $f(T_0)$ are as follows.

The first, based on the Clausius-Clapeyron equation, gives:

$$f_1(T_0) = \frac{L_{lv}}{R_v T_0^2} \exp\left(\frac{L_{lv}}{R_v} \left[\frac{1}{T_{\text{tr}}} - \frac{1}{T_0}\right]\right)$$

where L_{lv} is the latent heat of evaporation ($2.6 \cdot 10^6 \text{J/kg}$) and R_v is the specific gas constant for water vapor (461J/kg K).

The second is a polynomial approximation derived from observations:

$$f_2(T_0) = \sum_{n=2}^7 \frac{a_n}{a_1} (n-1) (T_0 - T_{\text{tr}})^{n-2}$$

a_1	6.11176750	a_2	0.443986062	a_3	$0.143053301 \cdot 10^{-1}$
a_4	$0.265027242 \cdot 10^{-3}$	a_5	$0.302246994 \cdot 10^{-5}$	a_6	$0.203886313 \cdot 10^{-7}$
a_7	$0.638780966 \cdot 10^{-10}$				

2.3 Precipitation

A physical model for precipitation involves $\mathcal{H}(r - r_p)$ where r is the relative humidity $\frac{q}{q_{\text{sat}}}$, r_p is some threshold value and \mathcal{H} is the Heavyside step function. This means it only rains when $r > r_p$.

These models are hard to linearize because of the discontinuous step function, and when linearized using a continuous function which approximates \mathcal{H} the coefficients contain $\mathcal{H}(r_0 - r_p)$, which is impossible to evaluate without data for the humidity distribution q_0 . Even if these measurements were available, it is not guaranteed that the balance of fresh water is closed, i.e. the integral of $E' - P'$ over the ocean's surface is 0.

Because of these problems a simpler model will be used. In this simpler model the precipitation will be taken as spatially homogeneous. The amount of precipitation can then be determined from the balance of fresh water:

$$\int E' - P' dA = 0$$

The equation for the precipitation then follows:

$$P' = \frac{1}{A_o} \int E' dA$$

where the integrals are over the ocean's surface and A_o is the ocean's surface area.

2.4 Scaling and summary of equations

Finally, some scaling factors are added to make the model compatible with the other unknowns in THCM, as well as making the quantities dimensionless. The evaporation and precipitation are scaled with $E' = U_o H \tilde{E} / r_0$, $P' = U_o H \tilde{P} / r_0$ where $r_0 = 6.4 \cdot 10^6 \text{m}$, $U_0 = 0.1 \text{m/s}$, $H = 5 \cdot 10^3 \text{m}$ are the radius of the earth, the velocity scale of the ocean circulation and a vertical length scale, respectively. Adding scaling with a specific humidity scale ($Q = 0.01$) and a temperature scale ($\Delta T = 1 \text{K}$) results in:

$$\tilde{\varepsilon}_T = \mu \tilde{U}_a(x) \tilde{f}(T_0) \quad (2.1)$$

$$\tilde{\varepsilon}_q = \mu \tilde{U}_a(x) \quad (2.2)$$

$$\tilde{E} = \tilde{\varepsilon}_T T' - \tilde{\varepsilon}_q q' \quad (2.3)$$

$$\tilde{P} = \frac{1}{A_o} \int \tilde{E} dA \quad (2.4)$$

$$\frac{\partial q'}{\partial t} = P_H^V \left(\frac{\partial^2 q'}{\partial x^2} + \delta \frac{\partial^2 q'}{\partial y^2} \right) + \nu_q (\tilde{E} - \tilde{P}) \quad (2.5)$$

$$\frac{\partial q'}{\partial n} = 0 \quad \text{at the boundaries} \quad (2.6)$$

Where:

μ	$= 1.6 \cdot 10^{-5}$	$\tilde{U}_a(x)$	$= U_a(x) / U_0$
P_H^V	$= \kappa_x / U_0 r_0$	$\tilde{f}(T_0)$	$= \epsilon e_{\text{sat, tr}} \Delta T f(T_0) / p Q$

One problem that can be seen from these equations is that the solution is not uniquely defined. If $\frac{\partial q'}{\partial t} = 0$ has a solution $\mathbf{q}' = \mathbf{a}$ then $\mathbf{a} + c(1 \dots 1)$ is also a solution. In other words, the solution vector \mathbf{q}' is only determined up to a constant.

This can be fixed by adding an extra constraint. The extra constraint that will be used here is $\int q' dA = 0$. Besides from being a natural way to make the solution unique, the discretized version of this constraint can be seen as forcing the solution vector for q' to be orthogonal to $(1 \dots 1)$, effectively making $c = 0$.

2.5 Influence on salinity

The model does not have any influence on the oceanic flow yet, this will be added by considering the influence of the evaporation and precipitation on the salinity in the upper layer of the ocean. This is modelled by adding the following term to the salinity equation:

$$\frac{\partial S}{\partial t} = \dots + \nu_s (\tilde{E} - \tilde{P})$$

for S in the upper layer of the ocean.

ν_s is determined from the characteristic sea-surface salinity, and is approximately 700.

Chapter 3

Discretization and Implementation

For the implementation of the addition of the vapor model, I will start by outlining some requirements. Then, some internals of THCM that are necessary to understand the implementation will be discussed. Finally, the most important parts of the discretization of the equations and the implementation of these will be presented, skipping most details as these can be found in the implementation itself (Appendix A).

3.1 Requirements

First, some requirements for the implementation will be created. These will be guidelines used throughout the implementation process and will ensure high quality software. The specific numbers in the efficiency requirement were determined by considering the highest grid resolution that was used up to now and taking a resolution substantially above this, so the efficiency will be sufficient for all future versions and experiments.

- The implementation should be mathematically correct.
- It should be compatible with the existing system.
- It should be modular, and easy to switch on or off.
- It should be maintainable and thoroughly commented.
- It should be efficient enough to handle grids with 360×90 atmospheric gridpoints, and situations where the main matrix can have a dimension of over 5,000,000.

3.2 Understanding THCM

Completely understanding THCM is far beyond the scope of this thesis, as it consists of many thousand of lines of code. It is also unnecessary as many parts, including the solver, can be considered a 'black box'.

The most important parts to understand are the parts where the Jacobian matrix is constructed. This is done by first constructing an array of local links and then constructing the matrix from this array.

Table 3.1: A1 offsets

below			same height			above		
12	15	18	3	6	9	21	24	27
11	14	17	2	5	8	20	23	26
10	13	16	1	4	7	19	22	25

3.2.1 The array of local coefficients A1

A1 is a six-dimensional array containing all the local links of the discretization. In this array it is possible to set influences between any two unknowns, in any two neighbouring grid points.

The statement $A1(i, j, k, Offset, Var, ByVar) = C$ would mean that the variable **Var** in the gridpoint (i, j, k) is influenced by the variable **ByVar** in the gridpoint $(i, j, k) + OffsetVector$.

OffsetVector can be looked up from the Offset number (1 – 27) in table 3.1. In this table 'below', 'same height' and above correspond to $-1, 0$ and 1 as third element in the OffsetVector respectively. The first and second element of the vector are determined likewise from the horizontal and vertical coordinates in the smaller tables.

In terms of equations, $A1(i, j, k, 14, A, B) = c$ would correspond to $\frac{\partial A}{\partial t}(i, j, k) = \dots + c B(i, j, k - 1)$ because the OffsetVector for the number 14 is $(0, 0, -1)$.

3.2.2 Compressed sparse row format

The entries in the array of local influences are then converted to a single large matrix. Because the equation for the precipitation is inherently non-local and can therefore not be stored in A1, the matrix will have to be changed afterwards.

This matrix is very sparse, so to save memory and CPU power it is stored in the so-called compressed sparse row format. Understanding this format will be essential for implementing the precipitation equations.

The storage format consists of:

- A scalar containing the dimension of the matrix.
- An array containing all the nonzero coefficients.
- An array containing all the column numbers for the coefficients.
- An array containing the indices for the coefficients and column number arrays at which the entries for a specific row start and end.

These four variables are named `ndim`, `coA`, `jcoA` and `begA` for the Jacobian in THCM.

Example:

$$\begin{pmatrix} -2 & 1 & 0 \\ 1 & -3 & 1 \\ 0 & 1 & -4 \end{pmatrix}$$

Would be represented as:

variable	value
<code>ndim</code>	3
<code>begA</code>	{1, 3, 6, 8}
<code>coA</code>	{-2, 1, 1,-3, 1, 1,-4}
<code>jcoA</code>	{ 1, 2, 1, 2, 3, 2, 3}

3.2.3 Grid

THCM uses a special type of grid, in which not all unknowns are taken at the same position in the grid cell.

Because there is no equation involving oceanic velocities in the water vapor model, it is sufficient to know that pressure, temperature and salinity are all taken as the value in the middle of the cell, so the grid can be considered a standard uniform 3D grid with values at the center.

3.3 Implementation

The only variables used in THCM are those for velocities (u, v, w) , pressure p , salinity S and temperature T . There is no specific room for the humidity q in the matrix and creating this would be very problematic and bug-prone. Of course there are several unused variables in the atmospheric layer, as only T is already used. Therefore we will use one of the other variables for q .

There are links from the temperature at sea level to the humidity, and from the humidity to the salinity. To keep all links within the S/T block of the matrix we choose to use the salinity S to represent the humidity q in the atmospheric layer. This keeps all entries as close together in the matrix as possible.

A constant signifying which variable is used for q is kept as a local parameter to allow this to be changed easily and to improve readability.

```
integer, parameter :: QQ = SS
```

3.3.1 Tools

The values of $\tilde{f}(T_0)$, $\tilde{\varepsilon}_q(x)$, $\tilde{\varepsilon}_T(x)$ need to be determined. This is just straightforward implementation of the formulas. Both f_1 and f_2 will be implemented, and a function `f_tilde` will be made so that the choice for f_1 or f_2 can be changed easily. This choice will be set to f_2 , as this is claimed to be slightly more accurate.

3.3.2 Diffusion

The diffusion equation contains a derivative which needs to be discretized. The starting point for this is the standard discretization for a second derivative on an equidistant grid:

$$\left. \frac{\partial^2 q'}{\partial x^2} \right|_{x_{i,j}} = \frac{q'_{i+1,j} + q'_{i-1,j} - 2q'_{i,j}}{h^2}$$

Several problems appear here: First, the grid distance h for the discretization of $\frac{\partial^2 q'}{\partial x^2}$ is not constant, but varies with the latitude. Also, the area of the cells involved vary with latitude which adds a scaling factor to the discretization of $\frac{\partial^2 q'}{\partial y^2}$.

To avoid these problems we make use of the fact that this derivative is already implemented in THCM for other variables, and we can simply copy these values. The values already generated for $\frac{\partial^2 T}{\partial x^2}$ will be used for $\frac{\partial^2 q'}{\partial x^2}$, this way correctness is guaranteed and duplication of code is avoided.

Altitude (and therefore grid distance) differences between layers are ignored by the discretization of $\frac{\partial^2 T}{\partial x^2}$. We will also ignore the altitude difference between ocean and atmosphere considering the altitude of the vapor distribution is much smaller than the radius of the Earth ($1.8 \cdot 10^3 \ll 6.4 \times 10^6$). This allows the use of the discretization of $\frac{\partial^2 T}{\partial x^2}$ in the uppermost layer of the ocean for the discretization of $\frac{\partial^2 q'}{\partial x^2}$ in the atmosphere.

```

real    txx(n,m,l,np) , tyy(n,m,l,np)
real    qxx(n,m,np)   , qyy(n,m,np)

call tderiv(3,txx)
call tderiv(4,tyy)
qxx = txx(:, :, l, :)
qyy = tyy(:, :, l, :)

Al(:, :, l+1, :, QQ, QQ) = P_HV * (qxx + delta * qyy)

```

Boundary conditions

As described earlier, on all four boundaries the Neumann boundary condition $\frac{\partial q'}{\partial n} = 0$ will be used.

Because the actual gridpoints for q are in the middle of the cubes, the boundary is in fact between two gridpoints.

Considering the boundary condition for the left edge and discretizing $\frac{\partial q'}{\partial t} = 0$ there with $q'_{0,j}$ as a virtual grid point gives $\frac{q'_{1,j} - q'_{0,j}}{h} = 0$, so $q'_{1,j} = q'_{0,j}$.

This can be implemented by adding the influence of the nonexistent grid point $q'_{0,j}$ on $q'_{1,j}$ to the influence of $q'_{1,j}$ on itself, and then setting it to zero.

```

Al(1, :, l+1, 5, QQ, QQ) = Al(1, :, l+1, 5, QQ, QQ) + Al(1, :, l+1, 2, QQ, QQ)
Al(1, :, l+1, 2, QQ, QQ) = 0

```


The other three boundary conditions are done likewise:

Right boundary:

$$\begin{aligned} A1(n, :, l+1, 5, QQ, QQ) &= A1(n, :, l+1, 5, QQ, QQ) + A1(n, :, l+1, 8, QQ, QQ) \\ A1(n, :, l+1, 8, QQ, QQ) &= 0 \end{aligned}$$

Bottom boundary:

$$\begin{aligned} A1(:, 1, l+1, 5, QQ, QQ) &= A1(:, 1, l+1, 5, QQ, QQ) + A1(:, 1, l+1, 4, QQ, QQ) \\ A1(:, 1, l+1, 4, QQ, QQ) &= 0 \end{aligned}$$

Top boundary:

$$\begin{aligned} A1(:, m, l+1, 5, QQ, QQ) &= A1(:, m, l+1, 5, QQ, QQ) + A1(:, m, l+1, 6, QQ, QQ) \\ A1(:, m, l+1, 6, QQ, QQ) &= 0 \end{aligned}$$

Because the atmosphere extends over land, there are no additional boundaries other than the four outer ones.

3.3.3 Evaporation

The implementation of the evaporation part consists of setting two local influence coefficients for each gridpoint. These coefficients are only set over oceanic cells, this is done by using the `landm` array which contains the topographic data.

```
! Only add coefficients on ocean cells
if(landm(i,j,l) == OCEAN) then
  ! Influence of evaporation on the humidity
  A1(i,j,l+1,5,QQ,QQ) = A1(i,j,l+1,5,QQ,QQ) - nu_q * eps_q(i,j)
  ! Humidity influenced by ocean temperature, one layer lower (pt 14 in stencil)
  A1(i,j,l+1,14,QQ,TT) = A1(i,j,l+1,14,QQ,TT) + nu_q * eps_T(i,j)
endif
```

3.3.4 Salinity

The implementation of the influence of the humidity on the ocean's salinity also consists of setting two local influence coefficients for each ocean gridpoint. These can be added in the same loop where the evaporation coefficients are set.

```
! Salinity is influenced by the humidity, one layer higher (pt 23 in stencil)
A1(i,j,l,23,SS,QQ) = A1(i,j,l,23,SS,QQ) - nu_S * eps_q(i,j)
! ...And by the ocean temperature
A1(i,j,l,5,SS,TT) = A1(i,j,l,5,SS,TT) + nu_S * eps_T(i,j)
```

3.3.5 Precipitation

For the discretization of the precipitation, the integral in the precipitation equation needs to be replaced by a sum.

$$\begin{aligned}
\tilde{P} &= \frac{1}{A_o} \int \tilde{E}(x) dA \\
&= \frac{1}{A_o} \int \tilde{\varepsilon}_T(x)T'(x) - \tilde{\varepsilon}_q(x)q'(x) dA \\
&= \frac{1}{A_o} \sum_{x_i} A(x_i) (\tilde{\varepsilon}_T(x_i)T'(x_i) - \tilde{\varepsilon}_q(x_i)q'(x_i))
\end{aligned}$$

Because the grid is on a part of a sphere, the areas of the grid cells are not all identical, and these areas need to be determined to calculate the integral. A function `cell_area` is added to do this.

The value of the homogeneous precipitation is influenced by the ocean temperature and atmospheric humidity in every grid point. So any value which depends on the precipitation would need $2nm$ extra entries in its row if this equation was implemented directly. Inserting these entries into every row of the $2nm$ rows (salinity and humidity in every grid point) would cause $4n^2m^2$ coefficients to be inserted. For a 128×128 grid this is over 10^9 and completely destroys the sparsity of the matrix.

An alternative is to create an extra row in the matrix representing $-P + \sum_{x_i} A(x_i)E(x_i) = 0$. The corresponding column can then be used to create a dependency on P in other equations. This way only $4nm + 1$ extra entries are needed.

The relevant part of the matrix then looks like:

$$\left[\begin{array}{cccccccc}
& & & & & & & -\nu_q \\
& & & & & & & \vdots \\
& & & & & & & -\nu_q \\
& & & & & & 0 & q_{1,1} \\
& & & & & & & T_{1,1} \\
& & & & & & & \vdots \\
& & & & & & 0 & T_{n,m} \\
& & & & & & -\nu_S & S_{1,1} \\
& & & & & & & \vdots \\
& & & & & & -\nu_S & S_{n,m} \\
& & & & & & & \tilde{P} \\
-\frac{A(x_{1,1})}{A_o} \tilde{\varepsilon}_q(x_{1,1}) & \dots & -\frac{A(x_{n,m})}{A_o} \tilde{\varepsilon}_q(x_{n,m}) & \frac{A(x_{1,1})}{A_o} \tilde{\varepsilon}_T(x_{1,1}) & \dots & \frac{A(x_{n,m})}{A_o} \tilde{\varepsilon}_T(x_{n,m}) & 0 & \dots & 0 & -1
\end{array} \right]$$

The coefficients in the extra column are only set for ocean grid points. This is because, as described earlier, the precipitation is taken as spatially homogeneous over the ocean and completely ignored over land. An alternative for this is to set A_o to the area of the entire discretized part of the globe instead of just the ocean's surface area. Then precipitation over a land grid point could be taken as extra precipitation over the nearest ocean grid point, effectively simulating the water flowing back into the ocean.

There are several ways this row can be inserted in the matrix, these are referred to as the 'v1', 'v2' and 'v3' precipitation implementations.

v1

The first method tried is to insert the row and column exactly as depicted above using one of the grid points of a second unused variable as the row and column number.

For inserting the column, we will need to consider the efficiency of an algorithm that does this. The straightforward algorithm, iterating through all the rows and inserting entries where necessary by shifting all subsequent elements one position to the right, is far too slow. Assuming the column entries are sorted, so we can check if a row is the next one to need an extra entry in $\mathcal{O}(1)$ time, the algorithm has a complexity of $\mathcal{O}(n_c n_m)$, where n_c is the amount of nonzero entries in the column to be inserted, and n_m is the total amount of entries in the coefficient array. We know that $n_c = 2nm$, and the value of n_m can be estimated from the space reserved for the coefficient array ($\#$ grid cells $\cdot 6 \cdot 6 \cdot 27$), which can be up to several hundred million for larger grids.

Clearly, this algorithm is far too slow. The required (over 10^{12}) operations for larger grids would probably dominate the running time.

A faster method is to do the insertions backwards. This is possible because the final position of the last element is known in advance and the shifts required for extra space can all be done at once. The required time for this algorithm is $\mathcal{O}(n_m)$, which is also the theoretical minimum complexity. The code for this can be seen in appendix A.1 line 249-301, and is used in all versions to insert the extra row and column.

Implementing this algorithm causes the solver to break down complaining of a 'singular matrix'. This can be explained by considering the fact that the matrix is indeed a singular matrix because the constraint $\int q' dA = 0$ was not added here, making the solution only determined up to a constant.

v2

Adding the aforementioned constraint will make the matrix no longer singular. To do this, the constraint first has to be discretized. This is easily done, discretizing $\int q dA$ as $\int q dA = \sum_{x_i} A(x_i)q(x_i)$.

This equation could be added directly, replacing the equation for one of the humidity grid points with this constraint. However, we go further and use it to completely eliminate one of the humidity grid points.

This way, all links are kept within the $T/S/q$ block of the matrix and this could prevent further problems with the solver. The formula used for the eliminated grid point q_s follows directly from the discretization of $\int q dA$:

$$A(x_s)q(x_s) = - \sum_{x_i \neq x_s} A(x_i)q(x_i)$$

The formula for q_s can then be used to rewrite the precipitation equation, eliminating q_s

completely from it:

$$\begin{aligned}
P &= \frac{1}{A_0} \sum_{x_i} A(x_i)(\varepsilon_T(x_i)T(x_i) - \varepsilon_q(x_i)q(x_i)) \\
&= \frac{1}{A_0} \left(A(x_s)(\varepsilon_T(x_s)T(x_s) - \varepsilon_q(x_s)q(x_s)) + \sum_{x_i \neq x_s} A(x_i)(\varepsilon_T(x_i)T(x_i) - \varepsilon_q(x_i)q(x_i)) \right) \\
&= \frac{1}{A_0} \left(A(x_s)\varepsilon_T(x_s)T(x_s) - \varepsilon_q(x_s) \left(- \sum_{x_i \neq x_s} A(x_i)q(x_i) \right) + \sum_{x_i \neq x_s} A(x_i)(\varepsilon_T(x_i)T(x_i) - \varepsilon_q(x_i)q(x_i)) \right) \\
&= \frac{1}{A_0} \left(A(x_s)(\varepsilon_T(x_s)T(x_s)) + \sum_{x_i \neq x_s} A(x_i)(\varepsilon_T(x_i)T(x_i) - (\varepsilon_q(x_i) - \varepsilon_q(x_s))q(x_i)) \right)
\end{aligned}$$

The point $q_{n,m}$ is used for q_s .

The algorithm now consists of the following steps:

- For all rows with an entry at the column for q_s , remove the entry and insert a small subrow corresponding to $-\sum_{x_i \neq x_s} A(x_i)q(x_i)$
- Insert the row and column for the new precipitation equation at the row/column previously used for q_s .

For inserting the row and column, the same method as with the v1 implementation can be used. For inserting the small rows to replace the entries in the column for q_s , a new function will have to be made. An algorithm which simply shifts all subsequent entries to the right to make room for the small row can be used here, as there are only 2 or 3 of such entries in total.

Implementing this algorithm fixes the problem with the solver.

v3

Another possibility to make one of the humidity grid points redundant is to assume $q_{n-1,m} = q_{n,m}$. The advantage of this method is that only a few matrix entries have to be inserted, instead of several rows of nm entries. However, the big disadvantage is that the constraint is that the constraint on $\int q dA$ is not added.

Instead of simply replacing every link to $q_{n,m}$ with one to $q_{n-1,m}$, the sum of the two cells is taken as the new value, effectively merging the two cells into one larger cell.

This also requires a few changes in the matrix: If $\frac{dq_{n,m}}{dt} = F_1$ and $\frac{dq_{n-1,m}}{dt} = F_2$ then $\frac{dq_{n,m}}{dt} + \frac{dq_{n-1,m}}{dt} = F_1 + F_2$. For the A matrix this corresponds to adding the row for $q_{n,m}$ to the row for $q_{n-1,m}$.

In addition to this change, any row corresponding to a variable $v_{i,j}$ that is influenced by $q_{n,m}$ or $q_{n-1,m}$ needs to have some coefficients changed.

If $\frac{dv_{i,j}}{dt} = G + aq_{n,m} + bq_{n-1,m}$ then, after merging the grid cells, $\frac{dv_{i,j}}{dt} = G + \frac{a+b}{2}(q_{n,m} + q_{n-1,m})$. For the A matrix this corresponds to adding the column for $q_{n,m}$ to the column for $q_{n-1,m}$ and then dividing the resulting value by 2. Also, the row and column for $q_{n,m}$ both have to be zeroed out to make room for the precipitation equation.

This method also causes no problems with the solver, leaving two working implementations of the precipitation equation.

The fact that the solver is able to solve this system is somewhat surprising, because the matrix is still singular. Although direct methods like Gaussian elimination would consistently fail on this system, iterative (Krylov subspace) methods like the one used in THCM are often still able to solve such a system.

Chapter 4

Visualization

To study the results and differences between several results, they will have to be visualized.

For the visualization, a tool will be made that can take a solution produced by THCM and produce plots of each variable at each height level in real-time.

4.1 Tools used

Ruby Ruby [4] is a multi-paradigm programming language, supporting a mix of object-oriented and functional programming styles. It allows for very quick prototyping. This is one of the main reasons it was chosen for this tool.

Tk Tk is a cross-platform, open source GUI toolkit. It has become the de-facto standard for building graphical user interfaces with Ruby, which is also the reason it was chosen for this project.

gnuplot gnuplot [5] is a stand-alone program that generates plots from functions or data input. It has support for a wide variety of plots including 2d, 3d, contour plots and vector fields. The program was chosen over other plotting programs for its ability to take text input and quickly generate a plot as an image file.

Ruby 'rgplot' package The Ruby 'rgplot' package [6] functions as an interface between Ruby and the gnuplot program. It uses a pipe to send text-based commands to a stand-alone gnuplot process. The license is a permissive open source license, similar to the MIT license, which allows for changing and redistributing the package.

4.2 Implementation

4.2.1 Reading data

To visualize the data, it first has to be read from a file. One way to do this is by using the `fort.3` file THCM uses as output. This file contains several input parameters, followed by some other parameters, followed by the data: value of each unknown in each grid point, its derivative and some other quantities. A problem with reading this file format is that the number of parameters is only known in the fortran code, and not stored in the file. Adding these to the Ruby script would require duplication of code.

Instead of doing this, code is added to THCM to generate an extra file after writing `fort.3`. This file just contains the grid dimensions and the data, making it very portable and easy to read without any duplicate definitions. Reading the data from this is accomplished with a standard construction of some nested loops.

To compare two different solutions, a second input file will be allowed whose data will be subtracted from the data of the first input file.

4.2.2 Graphical User Interface

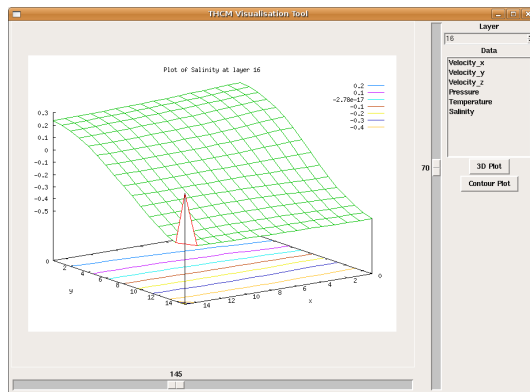
Next, a graphical user interface has to be designed and implemented.

The central part will be an area where the result can be shown, this is done with a `TkCanvas` widget. This widget is a generic canvas, an empty piece of the GUI with a variety of functions for drawing and display of images.

Next a `TkSpinbox` is added to allow the selection of a layer to plot, and a `TkListbox` is added for selecting a variable to plot : velocity (u, v, w), pressure (p), salinity (S) or temperature (T).

Two sliders are added to control the viewing angle of the 3D plot, and two buttons are added to allow for both 3D and contour plots.

All the widgets are put in frames to complete the layout, and some labels are added.



Finally, some keyboard bindings are added to allow the viewing angle and layer to be changed using keyboard commands.

Table 4.1: The keyboard commands for the visualization tool.

key	action
a	one layer up
z	one layer down
arrow keys	rotate 3D plot
enter	generate 3D plot
c	generate contour plot

4.2.3 Plotting

The `plotf` function called by the GUI code still has to be implemented.

This function retrieves the plotting parameters, and uses these to send the appropriate commands to the `rgplot` library. This library then gives a plot command to `gnuplot`, which saves the resulting plot in `tmp.gif`. The image file is then placed on the Tk canvas to show the result to the user.

Chapter 5

Results

In this chapter the new model will be tested, the performance will be measured and the results will be discussed.

Because data for the wind velocity $U_a(x)$ is currently not available, this chapter will focus on influence of this parameter on the results. The parameter will be varied between 1 and 10 to test the influence of the wind velocity on the vapor distribution and the salinity distribution.

First, some performance testing will be done to test the impact of the vapor model on the running time. Next, the two precipitation models will be compared and the best one will be used in subsequent tests, in which the influence of the wind velocity will be tested.

Finally, the various results will be discussed.

5.1 Setup

Before any results can be obtained, THCM first has to be configured. There are also some undetermined parameters of the vapor model left to choose.

In the vapor model the eddy diffusion κ_x, κ_y , the wind velocity $U_a(x)$ and the initial temperature T_0 are still unknown. The eddy diffusion κ_x, κ_y are both set to 10^6 , as suggested by [3]. T_0 is set to a constant $273.16 + 15$ which is the base temperature level in THCM. Also, as described in the implementation chapter, only the function f_2 will be used for the calculation of $\tilde{f}(T_0)$.

The rest of the model is configured as 'idealized', i.e. depending on a simplified model of wind forcing and other external factors, instead of depending on external data.

The salinity forcing is set to a restoring condition. With this option on the surface salinity is forced towards a known value. This will interfere with the influence the vapor model has on surface salinity, and possibly negate most effects on oceanic flow. However, the solvers are currently unable to solve the model without this restoring condition, making this option a necessity.

The geographical position of the part of the ocean that is modeled is the North Atlantic basin, the latitude running from 10 to 74 degrees N and the longitude from 10 to 74 degrees W. The topology is set to 'all ocean', without any continents.

The resolution is set to $16 \times 16 \times (16 + 1)$ grid cells. This makes each cell represent a 4 degree by 4 degree area on the surface of the globe. This guarantees at least some accuracy while converging within a reasonable amount of time on a desktop PC.

The solution is obtained by using a continuation process on the 'combined continuation' parameter in THCM. This parameter combines all the external forcing, so the solution process effectively goes from the known solution with no external forcing (the zero solution) to a stationary solution of the model with full external forcing.

5.2 Timings

For both of the working implementations, and for all wind speeds between 1 and 10, the time it takes to solve the model was measured.

U_a	v2 timing	v3 timing
off/0	1662.4	1662.4
1	1641.0	1600.1
2	d.n.c (0.23)	1784.1
3	1721.3	2060.2
4	1429.8	1447.6
5	1456.9	d.n.c (0.29)
6	1231.4	d.n.c (0.93)
7	1527.6	d.n.c (0.38)
8	885.4	1633.9
9	1199.9	1396.6
10	1997.5	1672.7

Table 5.1: Timing results for both implementations of the precipitation model and varying wind velocities. Time is in seconds and was measured using the Unix 'time' command.

d.n.c. (x) means that the Newton method did not converge at some point in the continuation process, with the continuation parameter being x (a run is completed when this parameter is 1). In this case the program halted with the message 'Newton not converged after six restarts'.

From these results, it is clear that the addition of the vapor model does not significantly influence the performance. Insofar it does, this effect is irregular and therefore probably a result of the matrix solver code, and not of the actual construction of the matrix.

5.3 Comparison

The following will have to be compared:

- The difference between the 'v2' and 'v3' precipitation implementations.
- The difference between the oceanic flow with the vapor model off and the vapor model on.
- The influence of the wind velocity on the vapor distribution and oceanic flow.

Of course these are far too many results to make a full pairwise comparison, so only a few of them will be chosen. First, the two precipitation implementations will be compared, and only the better one will be used in subsequent comparisons. Next, four different runs will be compared: "vapor model off", $U_a = 1$, $U_a = 4$ and $U_a = 8$. This allows for studying the influence of the wind velocity on the humidity distribution and oceanic flow without resulting in too many plots. With no wind at all ($U_a = 0$), there is no evaporation and the solution will be the same as with the vapor model turned off, so this is not tested explicitly.

To further limit the amount of plots, only the following plots will be made for each comparison:

- Atmospheric temperature and humidity.
- Salinity and temperature near the surface, because these layers are influenced by, or influence the humidity.
- Pressure and horizontal velocity near the surface, at the middle and near the bottom of the ocean. This makes it possible to study the influence of the humidity on the oceanic flow.

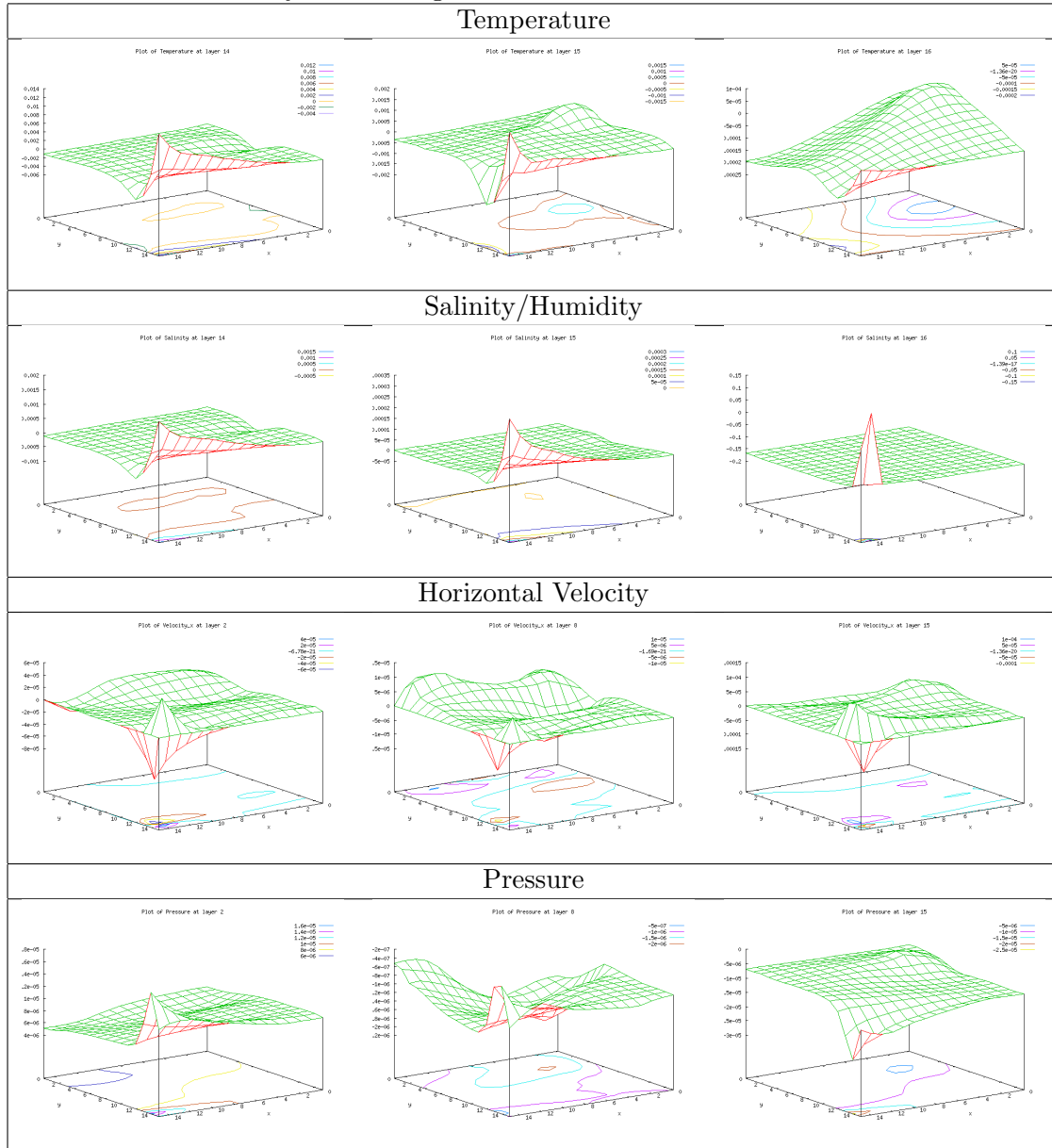
In all these plots the layer indices are zero-based, i.e. 'layer 0' is at the bottom and 'layer 15' at the surface of the ocean. Layer 16 is the atmospheric layer, and the salinity at this level is in fact the humidity, as described earlier.

The viewpoint of all plots is from the northeast, somewhere near iceland. The x coordinate increases from west to east and the y coordinate increases from south to north.

5.3.1 Comparison of v2 and v3 precipitation models

First, the two different precipitation models will be compared. The wind velocity will be set to $U_a = 1$ for this test:

Table 5.2: Some plots of the difference between the results of the v2 and v3 versions. These results were obtained by subtracting the v3 solution from the v2 solution



The two humidity distributions differ by a constant: the humidity in the v3 model is about 0.17 above the v2 model. This difference remains the same for different wind velocities, so it seems to be an inherent difference between the models. Although the fact that there is a constant difference between both results is easily explained by the lack of the extra constraint on the v3 model, the specific difference of 0.17 being constant over several different runs with

varying wind velocities remains unexplained.

There is no visible difference in the value of the precipitation, although this may be due to the fact that this value is very small.

The oceanic velocities, pressure and salinity differ only by about 10^{-5} to 10^{-4} , often without a discernable pattern. This might be largely caused by the errors in solving the systems for the continuation process. For higher wind velocities, these differences increase a bit, but only up to about 10^{-3} , and they are always at least an order of magnitude smaller than the values themselves.

Although there is not much difference, a choice has to be made between these to limit the amount of data to be compared when varying the wind velocity. The v2 model will be chosen, because:

- The value is $\int q$ is a much closer to 0. Although this value can not be determined in the v2 model (other than being 0 by definition), when using the value of $q_{n-1,m}$ for $q_{n,m}$ in calculating the integral the value is only about 0.001. This also shows the actual value of $q_{n,m}$ to be slightly lower than $q_{n-1,m}$, which is consistent with the rest of the solution, $q_{n,i}$ being slightly below $q_{n-1,i}$ at all other latitudes.

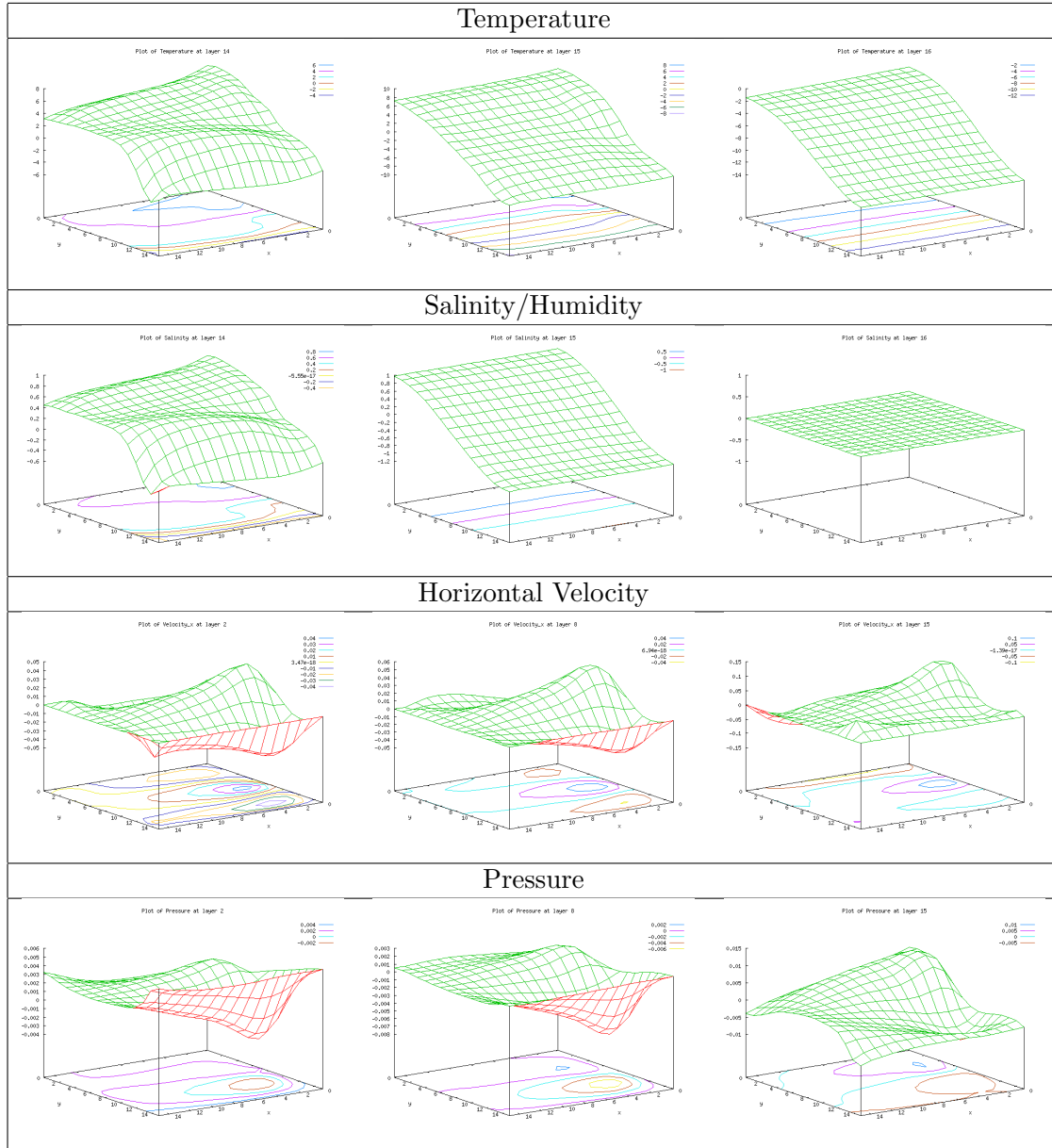
In the v3 model the integral is 0.15 when using a sphere with radius 1 for the calculation of the cell areas. This is because the v3 model does not include the constraint on $\int q$, making q only determined up to a constant. Also, no explanation for this specific constant could be found, making the model somewhat unpredictable.

- There is only one point in the humidity plots which doesn't fit (only $q_{n,m}$ instead of both $q_{n,m}$ and $q_{n-1,m}$).
- Contrary to expectations it also converges more often, and is faster than the v3 model.

5.3.2 Influence on oceanic flow

Results with vapor model off

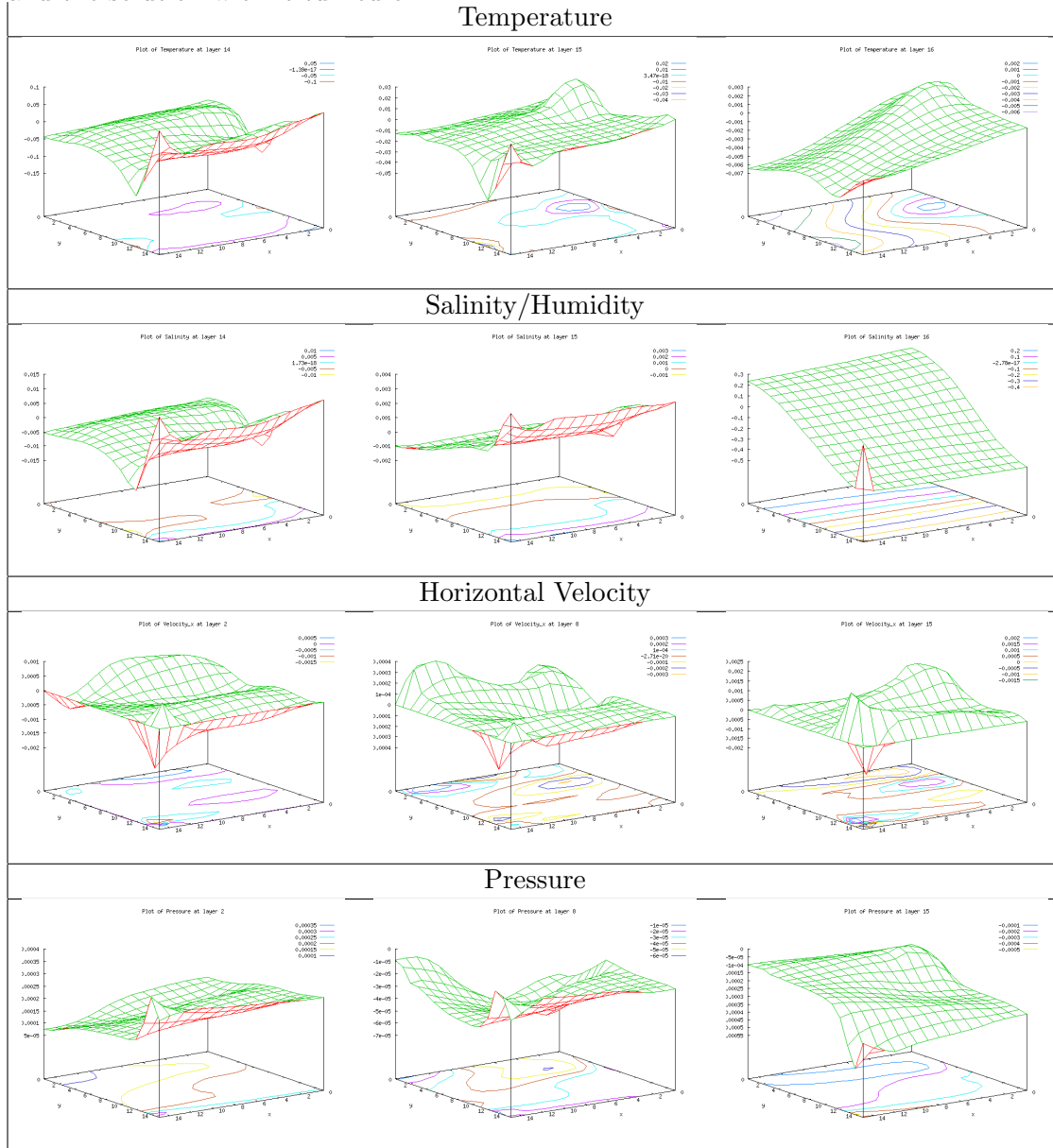
Table 5.3: Some plots of the solution with the vapor model turned off. This can be used as a baseline with which the other results can be compared.



All the other plots in this section are generated by taking the difference between the results with the vapor model on and the results with the vapor model off, because the plots are very similar and showing the results directly would hide most of the differences.

Wind velocity $U_a = 1$

Table 5.4: Some plots of the difference between the solution with the vapor model turned on and the solution with it turned off.



The humidity distribution runs from almost 0.3 near the equator to about -0.42 in the northern part. Also, the dependency on the sea surface temperature is very clear with the same diagonal contours appearing in both of these plots.

The difference in the sea surface salinity is only 10^{-3} , with the northern part having slightly higher salinity and the southern part a slightly lower salinity compared to the 'vapor model off' results. Although the effect is very small, this is unexpected because the part with

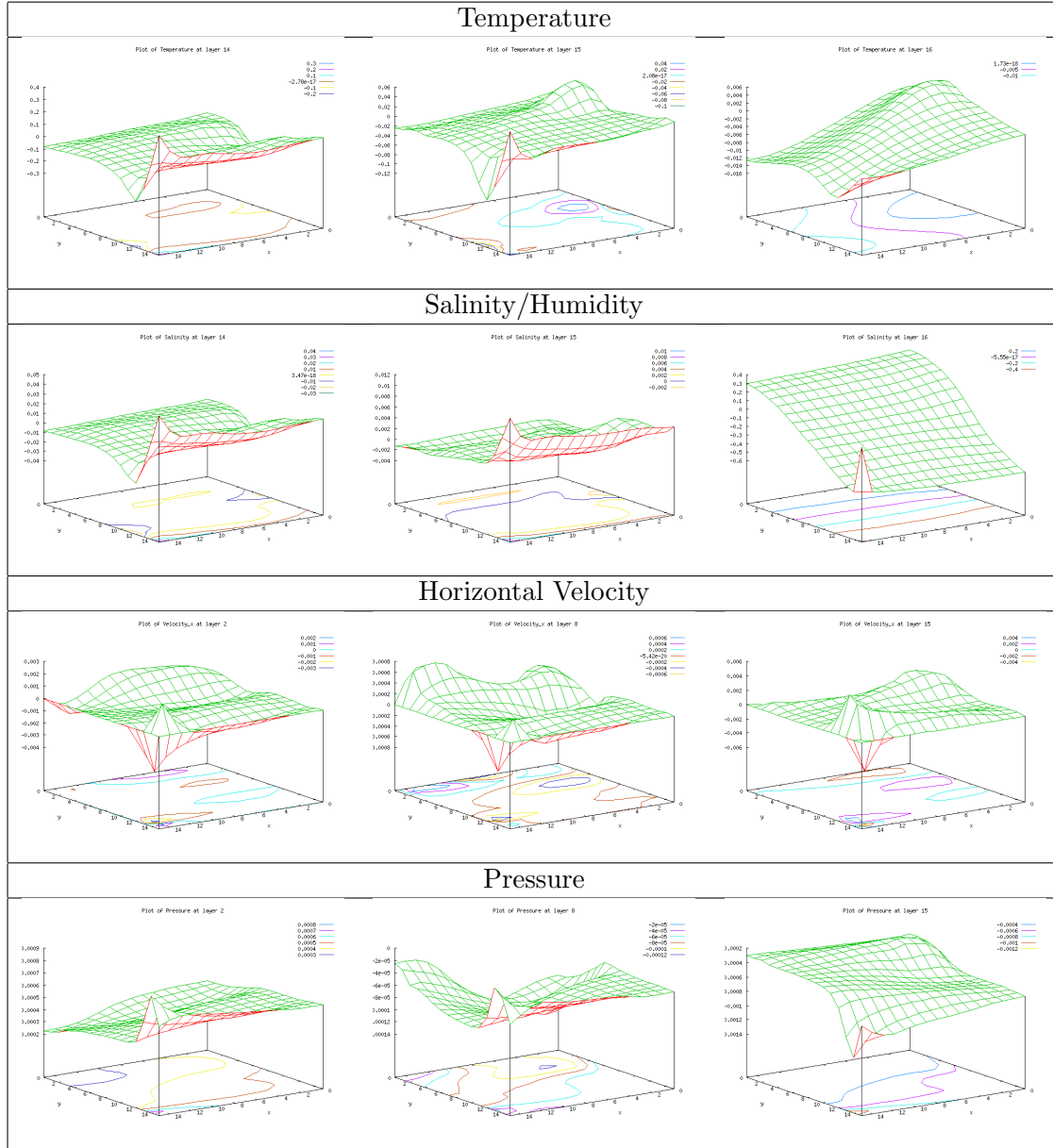
lower humidity is expected to have a lower $(E - P)$ value, resulting in a lower salinity.

There is almost no effect on oceanic flow: the difference in any of the velocities does not exceed 10^{-4} and the pressure difference is at about 10^{-5} . The difference in oceanic temperature is surprisingly large compared to the other differences, considering it is not directly influenced by the humidity in this model.

These strange results are probably the result of the restoring condition for salinity interfering with the influence the vapor model has on the surface salinity.

Wind velocity $U_a = 4$

Table 5.5: Some plots of the difference between the solution with the vapor model turned on and the solution with it turned off.

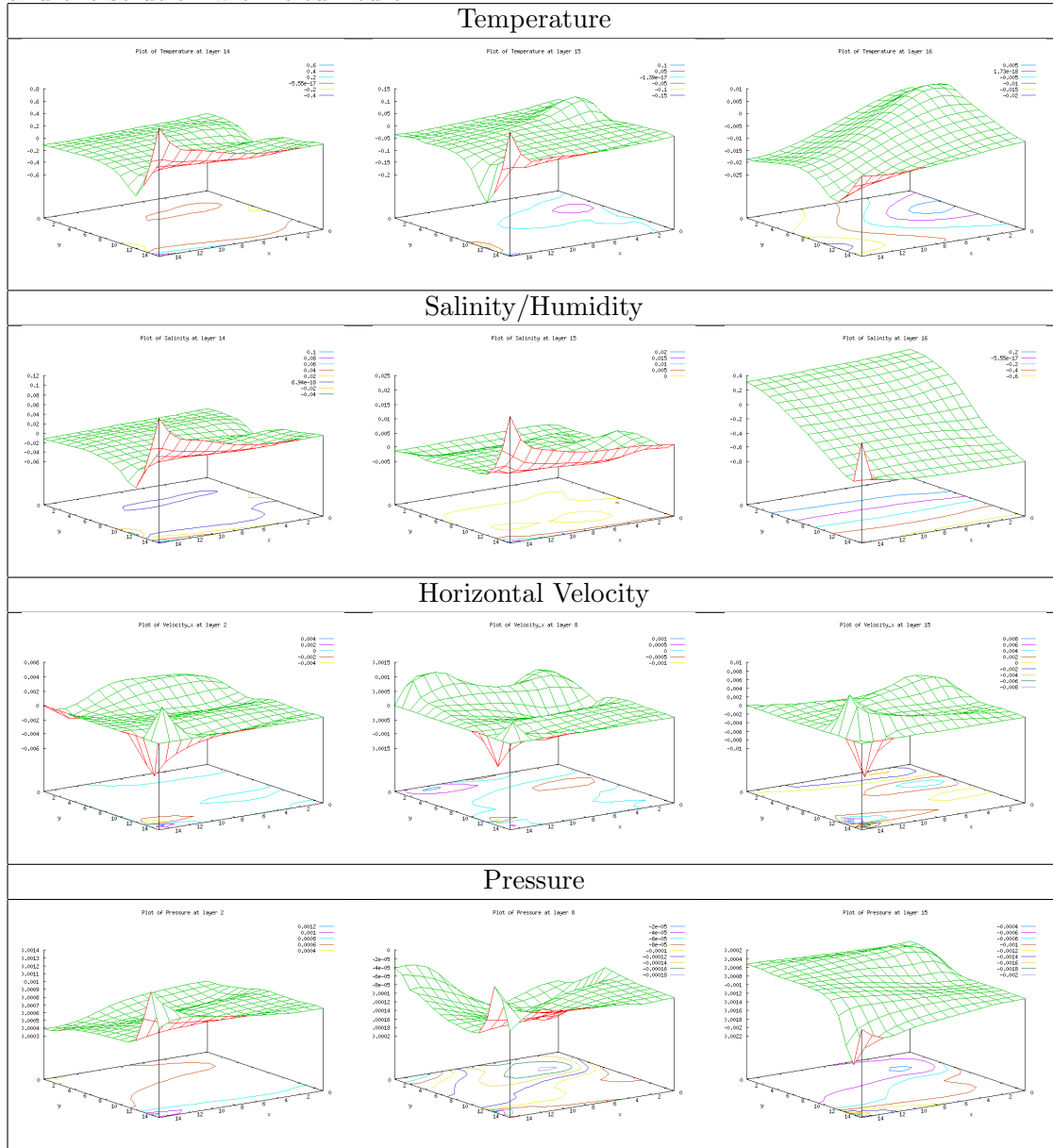


The maximum and minimum humidity values are considerably larger than in the $U_a = 1$ results.

The differences in velocity and pressure are a bit larger, at about 10^{-3} and 10^{-4} respectively. This is still a very small difference but it does show that the wind velocity has a fairly large influence on this difference, with a fourfold increase in wind velocity increasing these differences by a factor 10.

Wind velocity $U_a = 8$

Table 5.6: Some plots of the difference between the solution with the vapor model turned on and the solution with it turned off.



The humidity distribution here changes very little compared to the $U_a = 4$ case, and the other differences are also very similar to that case.

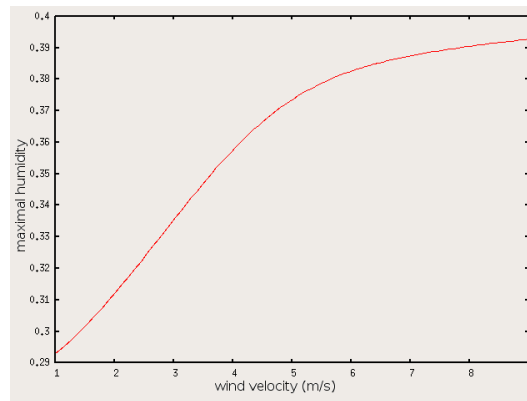
5.3.3 Effect of the wind velocity on the humidity distribution

Qualitatively speaking, the humidity distribution is roughly identical for all wind velocities. Only the magnitude differs in these cases, so the entire distribution can be represented by the maximum (or minimum) humidity value.

These maximum values are:

U_a	$\ q'\ _\infty$
1	0.2929
3	0.325
4	0.3736
5	0.38033
6	0.3849
7	0.38821
8	0.3907
9	0.39269

Plotting this, using bezier splines to make a smooth curve, results in:



This suggests a linear dependency on the wind velocity for the lower velocities, with the curve flattening as it nears saturation humidity.

5.4 Discussion

The vapor distribution met expectations, with the tropic regions having higher humidity than the polar regions. Also the dependency on wind velocity is consistent with the physical model.

The second version of the precipitation model turned out to be slightly more accurate and reliable, and this version is recommended as the version to permanently add to THCM. Timing both versions showed that neither one had a big impact on performance, although the solver did not always converge.

The difference in surface salinity between the 'vapor model off' results and the 'vapor model on' results is somewhat surprising. The expectation that there would be more rain on the more northern parts of the ocean, leading to less surface salinity, was not met. This is

probably because of the restoring condition for surface salinity. It is expected that this effect will change when the solvers are fixed and the restoring condition can be turned off.

However, there are several other possible sources of inaccuracies that may remain even if the problems with the solver are fixed. First of all, the initial temperature was set to a constant instead of being dependant on the actual oceanic surface temperature. This changes the magnitude of the evaporation.

Also, the model was configured as 'idealized' which may result in further inaccuracies, although these are likely to be merely quantitative changes. Finally, the homogeneous precipitation model could be inherently too inaccurate. This may be hard to change without destroying the fresh water balance, although adding some simple dependency on atmospheric temperature may still be possible. Also, the precipitation model could be adapted to include precipitation over land as described in section 3.3.5, which could further increase the accuracy.

Overall, the addition of the vapor model resulted in an efficient, maintainable and well-documented single module, which can now be used to extend the latest version of THCM.

Bibliography

- [1] Weijer, W. et al. *A fully-implicit model of the global ocean circulation* Journal of Computational Physics 192 452-470 (2003)
- [2] Weijer, W. *A vapor-anomaly model for THCM* (2001)
- [3] Weaver, A. J. et al. *The UVic Earth System Climate Model: Model Description, Climatology, and Applications to Past, Present and Future Climates* Atmosphere-Ocean 4, 361-428 (2001).
- [4] *Ruby programming language homepage* <http://www.ruby-lang.org/en/>
- [5] *Gnuplot homepage* <http://www.gnuplot.info/>
- [6] *Ruby Gnuplot homepage* <http://rgplot.rubyforge.org/>

Appendix A

Fortran code

A.1 v2

```
1 ! 2.x version
2
3 module m_vapor
4 use m_mat
5 implicit none
6 private
7 include 'usr.com'
8
9 !***** on/off switch *****
10 logical, parameter :: vapor_model_on = .true. ! set to false to disable the vapor model
11 public :: vapor_model_on
12
13 !***** physical constants and other parameters *****
14 integer, parameter :: QQ=SS ! use salinity in atmosphere to represent the humidity
15
16 real, parameter :: kappa_x = 1.0e6 ! eddy diffusivity
17 real, parameter :: delta = 1.0 ! kappa_y / kappa_x
18
19 real, parameter :: r0 = 6.4e6 ! radius of Earth
20 real, parameter :: U0 = 0.1 ! velocity scale
21
22 real, parameter :: P_HV = kappa_x / (r0*U0)
23
24
25 real, parameter :: nu_q = 2.2222e5
26 real, parameter :: nu_S = 700.0
27 real, parameter :: mu = 1.6e-5
28
29 real, parameter :: Llv = 2.5e6 ! Latent heat of evaporation (J/kg)
30 real, parameter :: Rv = 461.0 ! Specific gas constant for water vapor (J/kg K)
31 real, parameter :: Ttr = 273.16 ! Temperature at the triple point (K)
32 real, parameter :: esat_tr = 6.11 ! Vapor pressure at the triple point (hPa)
33 real, parameter :: atm_p = 1.0e3 ! Atmospheric pressure (hPa)
34 real, parameter :: eps_mix = 0.622 ! Mixing ratio of vapor and dry air (dimensionless)
35 real, parameter :: DltT = 1.0 ! Temperature scale (K)
36 real, parameter :: humsc_Q = 0.01 ! Humidity scale (kg/kg)
37
38
39 ! scaling factor for f: \tilde{f} = ftilde_factor * f
40 real, parameter :: ftilde_factor = eps_mix * esat_tr * DltT / (atm_p * humsc_Q)
41 ! coefficients for the polynomial approximation for f2
42 real, dimension(7), parameter :: ai = (/ 6.11176750 , &
43 0.443986062 , &
44 0.143053301e-1, &
45 0.265027242e-3, &
46 0.302246994e-5, &
47 0.203886313e-7, &
48 0.638780966e-10 /)
49
50 !***** private variables *****
51 real :: ocean_area
52 real, dimension(n,m) :: eps_q, eps_T
53 integer :: P_rownr ! Row number of precipitation row
54 integer :: P_rsize, P_csize ! # of nonzero row/column coefficients for precipitation
55 real, dimension(2*n*m+1) :: P_colc, P_rowc ! Coefficients for precipitation row/column
56 integer, dimension(2*n*m+1) :: P_col_i, P_row_j ! Indices for precipitation row/column
57
58 !***** public functions & parameters *****
59 public :: vapor_add.coeff, vapor_insert.precipitation_row, P_rownr
60
61 contains
```

```

62
63 ! Subroutine: vapor_add_coeff
64 ! -----
65 ! This function adds the local coefficients to the A matrix
66 ! Parameters:
67 ! un: the current solution vector -- This parameter is currently not used, but is passed to the
68 ! calc_epsilon_qT method, so the T_0 dependence can be made more accurate
69
70 subroutine vapor_add_coeff(un)
71 implicit none
72 ! -- Function parameters
73 real, intent(in), dimension(ndim) :: un
74 ! -- External functions
75 integer :: find_row2
76 ! -- Local variables
77 integer :: i, j, k, tempi
78 real :: tempr
79 logical :: flag
80
81 real, dimension(n,m,l,np) :: txx, tyy
82 real, dimension(n,m,np) :: qxx, qyy
83
84
85 ! Check if the vapor model is on, and if there is actually an atmospheric layer at all
86 if(vapor_model_on /= .true. .or. la /= 1) return
87 write(*,*) 'vapor> Vapor model is on, adding coefficients'
88
89 ! -- Initialize variables used here and in vapor_insert_precipitation_row
90
91 ! Calculate total area of the ocean
92 ocean_area = 0.0
93 do i=1,n
94   do j=1,m
95     if(landm(i,j,1) == OCEAN) ocean_area = ocean_area + cell_area(i,j)
96   enddo
97 enddo
98
99 ! Calculate the epsilon_q and epsilon_T coefficients
100 call calc_epsilon_qT(un)
101
102 ! ----- stencil -----
103 !   under   self   above
104 !  12 15 18   3 6 9   21 24 27
105 !  11 14 17   2 5 8   20 23 26
106 !  10 13 16   1 4 7   19 22 25
107
108 ! -- Add eddy-diffusive terms
109 call tderiv(3,txx)
110 call tderiv(4,tyy)
111 qxx = txx(:, :, 1, :)
112 qyy = tyy(:, :, 1, :)
113
114 A1(:, :, 1+1, :, QQ, QQ) = P_HV * (qxx + delta * qyy)
115
116 ! -- Add boundary conditions
117 ! Left boundary
118 A1(1, :, 1+1, 5, QQ, QQ) = A1(1, :, 1+1, 5, QQ, QQ) + A1(1, :, 1+1, 2, QQ, QQ)
119 A1(1, :, 1+1, 2, QQ, QQ) = 0
120
121 ! Right boundary
122 A1(n, :, 1+1, 5, QQ, QQ) = A1(n, :, 1+1, 5, QQ, QQ) + A1(n, :, 1+1, 8, QQ, QQ)
123 A1(n, :, 1+1, 8, QQ, QQ) = 0
124
125 ! Top boundary
126 A1(:, 1, 1+1, 5, QQ, QQ) = A1(:, 1, 1+1, 5, QQ, QQ) + A1(:, 1, 1+1, 4, QQ, QQ)
127 A1(:, 1, 1+1, 4, QQ, QQ) = 0
128
129 ! Bottom boundary
130 A1(:, m, 1+1, 5, QQ, QQ) = A1(:, m, 1+1, 5, QQ, QQ) + A1(:, m, 1+1, 6, QQ, QQ)
131 A1(:, m, 1+1, 6, QQ, QQ) = 0
132
133 ! -- Add evaporation term
134 do i=1,n
135   do j=1,m
136     ! Only add coefficients on ocean cells
137     if(landm(i,j,1) == OCEAN) then
138       ! Influence of evaporation on the humidity
139       A1(i,j,1+1,5,QQ,QQ) = A1(i,j,1+1,5,QQ,QQ) - nu_q * eps_q(i,j)
140       ! Humidity influenced by ocean temperature, one layer lower (pt 14 in stencil)
141       A1(i,j,1+1,14,QQ,TT) = A1(i,j,1+1,14,QQ,TT) + nu_q * eps_T(i,j)
142
143       ! Influence of evaporation on the salinity:
144       ! Salinity is influenced by the humidity, one layer higher (pt 23 in stencil)
145       A1(i,j,1,23,SS,QQ) = A1(i,j,1,23,SS,QQ) - nu_S * eps_q(i,j)
146       ! ...And by the ocean temperature
147       A1(i,j,1,5,SS,TT) = A1(i,j,1,5,SS,TT) + nu_S * eps_T(i,j)
148     endif
149   enddo
150 enddo
151

```

```

152  ! -- Construct row for precipitation, assuming \int q = 0
153
154  P_rownr = find_row2(n,m,1+1,QQ)
155  P_rsize = 1
156  do i=1,n
157    do j=1,m
158      P_rowc(P_rsize) = -(cell_area(i,j)/ocean_area) * ( eps.q(i,j) - eps.q(n,m) )
159      P_row_j(P_rsize) = find_row2(i,j,1+1,QQ)
160
161      ! Set the unused entry to 0, it will be removed by the packA function later on
162      if(find_row2(i,j,1+1,QQ) == P_rownr) P_rowc(P_rsize) = 0
163
164      P_rowc(P_rsize+1) = (cell_area(i,j)/ocean_area) * eps.T(i,j)
165      P_row_j(P_rsize+1) = find_row2(i,j,1,TT)
166      P_rsize = P_rsize + 2
167    enddo
168  enddo
169  P_rowc(P_rsize) = -1
170  P_row_j(P_rsize) = P_rownr
171
172  ! -- Construct column. Loop twice and in column-major order to get the array PP_rowno sorted.
173
174  ! Add influence of precipitation on ocean salinity
175  P_csize = 1
176  do j=1,m
177    do i=1,n
178      ! Only add for ocean cells
179      if(landm(i,j,1) == OCEAN) then
180        P_colc(P_csize) = -nu.S
181        P_col_i(P_csize) = find_row2(i,j,1,SS)
182        P_csize = P_csize + 1
183      endif
184    enddo
185  enddo
186
187  ! Add influence of precipitation on humidity
188  do j=1,m
189    do i=1,n
190      ! Only add for ocean cells, and don't add it for the removed humidity grid point q-n,m
191      if(landm(i,j,1) == OCEAN .and. find_row2(i,j,1+1,QQ) /= P_rownr ) then
192        P_colc(P_csize) = -nu.q
193        P_col_i(P_csize) = find_row2(i,j,1+1,QQ)
194        P_csize = P_csize + 1
195      endif
196    enddo
197  enddo
198  P_csize = P_csize - 1
199
200  ! -- Check if rows indices are sorted. This should be so, but row order might be changed later.
201  flag = .TRUE.
202  do i=1,P_csize-1
203    if(P_col_i(i) > P_col_i(i+1)) then
204      flag = .FALSE.
205      exit ! Break loop at first unsorted entry.
206    endif
207  enddo
208
209  ! Sort the indices array (and coefficients with it) if it is not sorted.
210  if(flag == .false.) then
211    write (*,*) 'vapor> Warning: Generated rows not in order, sorting to fix'
212
213    ! Selection sort algorithm
214    do i=1,P_csize-1
215      k = i
216      do j=i+1,P_csize
217        if(P_col_i(j) < P_col_i(k)) k = j
218      enddo
219      ! Swap values in both arrays
220      tempi = P_col_i(i)
221      tempr = P_colc(i)
222      P_col_i(i) = P_col_i(k)
223      P_colc(i) = P_colc(k)
224      P_col_i(k) = tempi
225      P_colc(k) = tempr
226    enddo
227  endif
228
229 end subroutine
230
231 ! Subroutine: vapor_insert_precipitation_row
232 ! -----
233 ! This function inserts the row and column for the precipitation equation matrix
234
235 subroutine vapor_insert_precipitation_row
236 implicit none
237 ! Local variables
238 integer :: ri, pci, i
239 integer :: coa.i.to, coa.i.from
240
241 ! Check if the vapor model is on, and if there is actually an atmospheric layer at all

```

```

242 if(P_rownr == 0 .or. vapor_model_on /= .true. .OR. la /= 1) return
243
244 ! First change rows affected by P_rownr (q-n,m) by using \sum A(x) q(x) = 0
245 call insert_qs_rows
246
247 pci = P_csize
248
249 ! Calculate final position of the current last element in coA/jcoA.
250 ! For inserting the row and column, P_csize + P_rsize extra space is needed.
251 coa.i.to = begA(ndim+1)-1 + P_rsize + P_csize - (begA(P_rownr+1) - begA(P_rownr))
252
253 ! Set the final begA array to this value+1 to indicate the last row will end just before this.
254 begA(ndim+1) = coa.i.to + 1
255
256 ! Check if there is enough space, and exit instead of causing segmentation faults.
257 ! If this error occurs, the coA/jcoA array sizes in assemble.f should be increased.
258 if(coa.i.to > size(coA)) then
259   stop 'Fatal error in vapor.F90: Not enough space to insert precipitation row and column. Increase coA/jcoA size'
260 endif
261
262 ! Insert the row and column by looping backwards
263 do ri=ndim,1,-1
264   ! Need to insert the row?
265   if(ri == P_rownr) then
266     ! Copy row to be inserted, don't copy entries already there.
267     do i=1,P_rsize
268       jcoA(coa.i.to) = P_row_j(i)
269       coA(coa.i.to) = P_rowc(i)
270       coa.i.to = coa.i.to - 1
271     enddo
272     coa.i.from = begA(P_rownr)-1
273   else
274     ! If not inserting a row, just copy the original row
275     do while(coa.i.from >= begA(ri))
276       coA(coa.i.to) = coA(coa.i.from)
277       jcoA(coa.i.to) = jcoA(coa.i.from)
278       coa.i.to = coa.i.to - 1
279       coa.i.from = coa.i.from - 1
280     enddo
281   endif
282
283   ! Check if a column entry needs to be inserted
284   ! A simple == check is possible because P_col_i is sorted
285   if(pci > 0) then ! Don't access P_col_i(-1) when entire column is already inserted
286     if(P_col_i(pci) == ri) then
287       jcoA(coa.i.to) = P_rownr
288       coA(coa.i.to) = P_colc(pci)
289       coa.i.to = coa.i.to - 1
290       pci = pci - 1
291     endif
292   endif
293
294   begA(ri) = coa.i.to + 1
295 enddo
296
297 ! Extra check to see if all the space was used.
298 if(coa.i.to /= 0) then
299   write(*,*) 'vapor> Error: inserting row and column failed: to-pointer = ',coa.i.to,'/= 0'
300   stop
301 endif
302
303 end subroutine
304
305 ! Subroutine: insert_qs_rows
306 ! -----
307 ! This function replaces all dependencies on q-n,m by a small row according to \sum A(x) q(x) = 0
308
309 subroutine insert_qs_rows
310   implicit none
311   ! External functions
312   integer :: find_row2
313   ! Local variables
314   integer :: i, j, ci, cj, ins, colj, sh
315
316   ! The shift needed for each row
317   sh = n * m - 2
318
319   do i=1,ndim ! For each row
320     do j=begA(i),begA(i+1)-1 ! For each entry in the row
321
322       ! If there is a non-zero dependency on P_rownr
323       if(jcoA(j) == P_rownr .and. i /= P_rownr .and. abs(coA(j)) > 1e-14) then
324
325         ! Shift all successive entries to the right
326         do cj=begA(ndim+1)-1,j+1,-1
327           coA(cj+sh) = coA(cj)
328           jcoA(cj+sh) = jcoA(cj)
329         enddo
330
331         do ci=i+1,ndim+1

```

```

332     begA(ci) = begA(ci)+sh
333     enddo
334
335     ! Replace the single coefficient by a small row
336     ins = 0
337
338     do ci=1,n
339         do cj=1,m
340             colj=find_row2(ci,cj,l+1,QQ)
341             if(colj .ne. P.rownr) then
342                 coA(j+ins) = -(cell_area(ci,cj)/cell_area(n,m)) * coA(j)  !- nulplossing
343                 jcoA(j+ins) = colj
344                 ins = ins+1
345             endif
346         enddo
347     enddo
348
349     if(ins /= sh+1) stop 'vapor.F90 error: insert.qs.rows: sh+1 .ne. ins'! Extra check.
350
351     ! Break the "j" loop: there can be only one such entry in each row
352     exit
353     endif ! If found entry
354     enddo ! For all entries in row
355     enddo ! For all rows.
356
357     ! Extra check. If this fires there are things already overwritten.
358     if(begA(ndim+1) > size(coA)) stop 'vapor> Not enough space to insert q rows. Increase coA/jcoA size'
359
360     ! Remove double entries by sort+merge
361     call sort_rows_A
362     call merge_doubles_A
363 end subroutine
364
365
366 ! Subroutine: sort_rows_A
367 ! -----
368 ! This function sorts the entries in each row by their column number
369
370 subroutine sort_rows_A
371   implicit none
372   integer :: i, j, k, cm, tempi
373   real :: tempr
374
375   do i=1,ndim ! For each row
376       ! Sort row: selection sort
377       do j=begA(i),begA(i+1)-1
378           cm = j
379           do k=j+1,begA(i+1)-1
380               if(jcoA(k) < jcoA(cm)) cm = k
381           enddo
382           tempi = jcoA(j)
383           tempr = coA(j)
384           jcoA(j) = jcoA(cm)
385           coA(j) = coA(cm)
386           jcoA(cm) = tempi
387           coA(cm) = tempr
388       enddo
389   enddo
390
391 end subroutine
392
393 ! Subroutine: merge_doubles_A
394 ! -----
395 ! This function merges column entries by adding them
396 ! if a row contains multiple entries for the same column
397 subroutine merge_doubles_A
398   implicit none
399   integer :: cf, ct, i, j, lc, cc
400   real :: tr
401
402   cf = 1 ! next entry to read
403   ct = 0 ! last position written
404   cc = 0 ! cleared count
405
406   ! For each row
407   do i=1,ndim
408       lc = -1 ! Clear last column #
409       do cf=begA(i),begA(i+1)-1 ! For all entries in this row
410           j = cf
411           ! Current column # is equal to the last, add them.
412           if(jcoA(j) > 0 .and. jcoA(j) == lc) then
413               coA(ct) = coA(ct) + coA(j)
414               cc = cc+1
415           else ! Not equal, just copy
416               ct = ct+1
417               coA(ct) = coA(j)
418               jcoA(ct) = jcoA(j)
419           endif
420           lc = jcoA(j)
421       enddo
422   enddo

```

```

422     begA(i+1) = ct + 1
423     enddo
424
425     write(*,*) 'vapor> Cleared ',cc,' doubles'
426 end subroutine
427
428 ! Function: cell_area
429 ! -----
430 ! This function calculates the area of the cell for grid point (i,j) on a sphere with r=1
431 real function cell_area(i,j)
432 implicit none
433 ! Parameters
434 integer, intent(in) :: i,j
435 ! Local variables
436 real :: latmin, latmax, ht
437
438 latmin = ymin + (j-1) * (ymax-ymin)/m ! Latitude of the bottom of the cell
439 latmax = ymin + j * (ymax-ymin)/m ! Latitude of the top of the cell
440 ht = (sin(latmax)-sin(latmin)) ! Height of the cell.
441
442 ! The slice of height ht has area 2 pi ht, multiply by % used in longitude for area
443 cell_area = 2 * pi * ht * ((xmax-xmin)/(2*pi))/n
444 end function
445
446 ! Function: f1, f2
447 ! -----
448 ! These are the two alternatives for the "f" function.
449 real function f1(T0)
450 implicit none
451 real, intent(in) :: T0
452 f1 = Llv/(Rv*(T0**2))*exp((Llv/Rv)*((1/Ttr)-(1/T0)))
453 end function
454
455 real function f2(T0)
456 implicit none
457 real, intent(in) :: T0
458
459 integer :: n
460 real :: tot
461
462 tot = 0.0
463 do n=2,7
464     tot = tot + (ai(n)/ai(1))*(n-1)*((T0-Ttr)**(n-2))
465 enddo
466 f2 = tot
467 end function
468
469 ! Function: ftilde
470 ! -----
471 ! \tilde{f} function used in calc_epsilon_qT, change choice of f1/f2 here
472 real function ftilde(T0)
473 implicit none
474 real :: T0
475 ftilde = ftilde_factor * f2(T0)
476 end function
477
478 ! Subroutine: calc_epsilon_qT
479 ! -----
480 ! Parameters:
481 ! un: the current solution vector.
482 subroutine calc_epsilon_qT(un)
483 implicit none
484 ! Parameters
485 real, intent(in), dimension(ndim) :: un
486 ! External functions
487 integer :: find_row2
488 ! Local variables
489 real, dimension(n,m) :: T0, Ua
490 integer :: i,j
491
492 T0 = Ttr + 15 ! This should eventually be changed to depend on un
493 Ua = 2 ! Scalar wind velocity, should be changed to depend on measurements or better estimates
494
495 Ua = Ua/U0 ! Ua tilde
496 do i=1,n
497     do j=1,m
498         eps_q(i,j) = mu * Ua(i,j)
499         eps_T(i,j) = eps_q(i,j) * ftilde(T0(i,j))
500     enddo
501 enddo
502
503 end subroutine
504
505
506 end module m_vapor

```

A.2 v3

Code not shown is identical to the previous listing.

```

1  ! 3.x version
2
3  module m.vapor
...
62 contains
...
71 subroutine vapor_add_coeff(un)
...
159     P_rowc( P_rsize) = -(cell_area(i,j)/ocean_area) * eps_q(i,j)
...
230 end subroutine
...
236 subroutine vapor_insert_precipitation_row
...
246 call merge_cells(next.to.P_rownr ,P_rownr)
247
...
304 end subroutine
305
306 ! Subroutine: merge_cells
307 ! -----
308 ! This function merges 2 grid cells by manipulating the matrix
309 subroutine merge_cells(cell_to,cell_from)
310 integer cell_to,cell_from,i,j,k, n,sh, toi, br, er
311 logical flag
312
313 ! columns: merge and divide by 2
314 do i=1,ndim
315 do j=begA(i),begA(i+1)-1
316 if(jcoA(j) .eq. cell_from) then           ! already an entry for cell_to ?
317 flag=.false.
318 do k=begA(i),begA(i+1)-1
319 if(jcoA(k) .eq. cell_to) then
320 coA(k)=coA(k)+coA(j)           ! ... then merge
321 coA(j)=0; jcoA(j)=0
322 flag=.true.
323 exit! break k loop
324 endif
325 enddo
326 if(.not. flag) jcoA(j)=cell_to ! ... else replace column # to create one
327 exit ! break j loop, next i
328 endif
329 enddo
330
331 ! divide entries by 2, cell_from = (cell_to+cell_from)/2
332 do j=begA(i),begA(i+1)-1
333 if(jcoA(j) .eq. cell_to) coA(j)=coA(j)/2
334 enddo
335 enddo
336
337 sh = begA(cell_from+1)-begA(cell_from)
338
339 br = begA(cell_to+1)
340 er = begA(ndim+1)-1
341 coA( br+sh:er+sh) = coA(br:er) ! shift
342 jcoA(br+sh:er+sh) = jcoA(br:er)
343
344 toi=begA(cell_to+1) ! start of new space in cell_to, clear
345 coA(toi:toi+sh-1) = 0
346 jcoA(toi:toi+sh-1) = 0
347
348 begA(cell_to+1:ndim+1) = begA(cell_to+1:ndim+1) + sh
349
350 do i=begA(cell_from),begA(cell_from+1)-1
351 flag=.false.
352 do j=begA(cell_to),begA(cell_to+1)-1
353 if(jcoA(i) .eq. jcoA(j)) then
354 flag=.true.
355 coA(j)=coA(j)+coA(i)
356 exit
357 endif
358 enddo
359 if(.not. flag) then
360 jcoA(toi)=jcoA(i)
361 coA(toi) =coA(i)
362 toi=toi+1
363 endif
364 enddo
365 coA(begA(cell_from):begA(cell_from+1)-1)=0
366 jcoA(begA(cell_from):begA(cell_from+1)-1)=0
367 end subroutine
...
447 end module m.vapor

```


Appendix B

Ruby Code

B.1 Visualization Tool

```
%w[gnuplot matrix scanf tk].each{|f| require f}

UU,VV,WW,PP,TT,SS = *1..6
Var_names = %w[ Velocity_x Velocity_y Velocity_z Pressure Temperature Salinity]

def read_data(file)
  data = Array.new(7)
  n,m,l,la,nun = *[]
  File.open(file,"r") {|f|
    n,m,l,la,nun = f.scanf("%d"*5)
    f.gets # clear newline
    nun.times{
      s=f.gets
      v = s.to_i # what variable
      puts Var_names[v] # should be each of the Var_names in order
      data[v] = []
      (l+la).times{|k|
        layer = []
        m.times{
          layer << (0..n).map{ f.gets.to_f }
        }
        data[v] << layer
      }
    }
  }
  [data,n,m,(l+la)]
end

solution,n,m,tl = read_data(ARGV[0]||"fort.1415")
if ARGV[1]
  solution2, = read_data(ARGV[1])
  for x in (1..6) ; for i in (0..n) ; for j in (0..m) ; for k in (0..tl)
    solution[x][k][j][i] -= solution2[x][k][j][i]
  end;end;end;end
end

# ----- build gui -----
root = TkRoot.new() { title "THCM Visualisation Tool" }
right = TkFrame.new(root).pack("side"=>"right", 'fill'=>'y', 'expand'=>true)
left = TkFrame.new(root).pack("side"=>"left", "fill"=>"both", 'expand'=>true)

x_rot_slider = TkScale.new(left) {
  from 0; to 180; set 70
}.pack('side'=>'right', 'fill'=>'y', 'expand'=>true)

TkLabel.new(right){text 'Layer' }.pack
layer = TkSpinbox.new(right){ from 0; to tl-1; set tl-1}.pack
TkLabel.new(right){text 'Data' }.pack
varsel = TkListbox.new(right){
  (1..6).each{|i| insert :end, Var_names[i] }
}.pack()

canvas = TkCanvas.new(left){
  width 700; height 600
}.pack('fill'=>'both', 'side'=>'top', 'expand'=>true)

z_rot_slider = TkScale.new(left) {
  orient 'horizontal'; from 0; to 360; set 145
}.pack('fill'=>'x')
```

```

# ----- plot function -----
plot_function = proc {|*options|
  plv = (varsel.cursorselection[0]||((SS-1))+1)
  Gnuplot.open do |gp|
    Gnuplot::SPlot.new( gp ) do |plot|
      plot.term 'gif'
      plot.out  '\tmp.gif\'

      plot.title  "Plot of #{Var_names[plv]} at layer #{layer.get}"
      plot.xrange "[0:#{n-1}]"
      plot.yrange "[0:#{m-1}]"
      plot.xlabel "x"
      plot.ylabel "y"

      plot.hidden3d
      if options[0] # contour plot
        plot.nosurface
        plot.view '0,0,1.2'
      else
        plot.view "#{x_rot_slider.get},#{z_rot_slider.get},1.15"
      end
      plot.contour 'base'

      plot.cnttparam 'levels 8'

      plot.data << Gnuplot::DataSet.new(Matrix[*solution[plv][layer.get.to_i]]) do |ds|
        ds.with = "lines"
        ds.notitle
      end
    end
  end
end

TkImage.new(canvas, 350, 300, 'image' => TkPhotoImage.new('file' => 'tmp.gif'))
}
# ----- more gui building -----
button = TkButton.new(right) {
  text "3D Plot"
  command proc{ plot_function.call }
}.pack

button = TkButton.new(right) {
  text "Contour Plot"
  command proc{ plot_function.call(1) }
}.pack

dturn = 15
root.bind('Key-Down') { x_rot_slider.set x_rot_slider.get+dturn; plot_function.call }
root.bind('Key-Up')   { x_rot_slider.set x_rot_slider.get-dturn; plot_function.call }
root.bind('Key-Left') { z_rot_slider.set z_rot_slider.get-dturn; plot_function.call }
root.bind('Key-Right') { z_rot_slider.set z_rot_slider.get+dturn; plot_function.call }
root.bind('a')        { layer.set [tl-1,layer.get.to_i+1].min ; plot_function.call }
root.bind('z')        { layer.set [0 ,layer.get.to_i-1].max ; plot_function.call }
root.bind('c')        { plot_function.call(1) }
root.bind('Return')  { plot_function.call }
plot_function.call

Tk.mainloop

```