Weather, Its Model, and Its Approximations

Jan Stepinski

CEE236B
June 8, 2015

Contents

1. Introduction

Weather is the solution of two equations: state and transport. For weather is the collection of properties of the fluid that is the atmosphere – namely density, pressure, and temperature – and the equation of state describes how they are, while the transport equation describes how they change along the wind. But this is not a well-posed problem; it is an abstraction. The problem is an abstraction because it evokes the mutual reinforcement of state properties and transport – with pressure, for example, driving wind which changes pressure – but it does not actually state what is being transported. Not densities, pressures, or temperatures. A warm day arises because of the transport not of temperature but rather warm air into a region. The concepts holds for density and pressure; the state properties arise because of the transport not of the property but of the air itself. The distinction is formally made between intrinsic and extensive properties. State variables are intrinsic in that they do not depend on the amount of air. Consider two regions of equal temperature. If they are merged, or if air is allowed to flow freely between them, the newly unified region does not become warmer; it simply has more air than the individual regions. That is the extensive property: the amount, the quantity of air. Intensive properties are certainly quantifiable, but they are not physical quantities of *stuff* that can be transported; they are descriptive measures of those quantities, and they change when the quantities are transported. So what exactly are these quantities? They are quantities of air. But to be mathematically tractable, they are separated into mass, energy, and momentum. Transport of the three is essentially identical. It is a conservation law, wherein the change of the quantity over time is equal to the net influx plus any net sources, and the flux is driven by spatial gradients as well as advection, or bulk flow along air currents. But despite the mathematical similarities, the underlying physical processes are unique – energy transport is derived from the laws of thermodynamics while momentum transport is derived from Newtonian mechanics, and turbulence arises from convection due to temperature gradients in the energy equation but from shear stresses due to density gradients in the momentum equation. And that also suggests why separating the transport of a quantity of air into the transport of quantities of mass, energy, and momentum is so useful: the latter three are determined by the three state variables, and wind. Thus the transport equation becomes a system of three equations governing four properties. And with the fourth equation being that of state, the problem becomes solvable. It generally cannot be deemed well-posed with information about its initial and boundary conditions, but with those set, it can be solved for the state properties, and thus it can be solved for the weather.

This study presents a solution of a coupled system of transport and state equations of the atmosphere, but it does not solve *weather*; it solves a mathematical representation of weather. A very simple one, where the air is treated as a homogenous mixture governed by the ideal gas law. But air consists of many gases, aerosols, and particulates, and a more realistic model will solve for the state of each one. It will transport each one, and in doing so it will solve for other aspects of weather, such as where vapor collects into rain. It will incorporate chemical reactions between these constituents, and reactions between them and external fluxes of energy in the form of solar radiation and mass in the form of pollution, and it will incorporate deviations in these fluxes caused by solar cycles and power plant operation cycles. And it will remain a model, elegant in its complexity, in its ability to extend the reach of seemingly unrelated distant variables to weather, but limited by the finitude of its reach. For weather is a natural phenomenon, and nature, by the nature of its unknowability, is infinite. It may be finite, but it may be the result of infinitesimal or cosmic perturbations – the proverbial butterfly rousing a hurricane – and because the reality is unknown, because there is no equation that can precisely replicate the eddies in a gust of wind, nature might as well be infinite. Any weather model is an approximation of this infinitude in a finite space of equations. But the equations exist in their own infinite mathematical spaces. A state property is a solution of a transport equation derived from a conservation law, so it is at least a piecewise continuous functional. And the algebraic equivalent of a functional is an infinite-dimensional vector. With an analytic form, such a functional would not pose an issue. It would simply be a formula for weather at any coordinate and any time, and its infinitude would be a byproduct of it being a mathematical structure. But a general model, even the simple model of this study, lacks an analytic solution. The solution – if one exists – must be approximated numerically, that is, as a finite-dimensional vector spanning a finite-dimensional domain of space and time. So not only is the model a finite approximation of an infinite nature, its “solution” is a finite approximation of an infinite vector space.

This study explores the accuracy and stability of the Matsuno scheme for solving the simple model of a homogeneous atmosphere with no turbulence and no external sources of mass or energy. Both of these metrics – accuracy and stability – depend on the discretization of the model. The discretization is the aforementioned finite-dimensional approximation of the domain;

it consists of a time step and three spatial steps, and the model is evaluated at nodes reached with a step in each dimension. If the steps are infinitely small, the resulting solution is true, in the mathematical sense. Of course they must be finite, so there is an error between the solution and the theoretical truth. An accurate solution scheme produces small errors relative to the step. For example, if halving the step halves the error, the scheme is accurate of order 1; if halving the step reduces the error by a factor of four, the scheme is accurate of order 2. Order has a precise mathematical definition, but its meaning is obvious by induction. A stable scheme preserves accuracy over time. A basic finite difference scheme is accurate of order 2 in space and 1 in time, but it is unstable. After several steps in time, the numbers begin to oscillate in sign and grow in magnitude, and they continue to grow until they can no longer be represented by the bits of a computer and the solution collapses into an array of nonsense. The Matsuno scheme retains the spatial differencing algorithm, but it repeats it twice within each time step, first evaluating an estimate of the value at the next time node, then adjusting it. The result is *conditional* stability. As long as the time step is small enough that wind cannot transport air over an entire spatial node, the solution will be stable. But there is also a lower bound on the time step: the computer. An arbitrarily small time step, say a nanosecond, will certainly result in a stable weather prediction – over a few minutes at most. For billions of iterations – and twice as many with the Matsuno scheme – of a highly complex algorithm would require an unfeasible amount of processing time and power, and the result would still be over a very short time for what is fundamentally a long-term process. A practical algorithm will minimize computational complexity and use the largest possible time step. This study begins by discussing an efficient implementation of the Matsuno time-stepping scheme with spatial finite-differencing in Fortran90. It then determines the largest stable time step and accuracy for its simple weather model and the Matsuno scheme. True accuracy is unknown because the true solution is unknown, but its order of magnitude can be determined by comparing the results of models with difference discretizations. Finally, the study introduces a single perturbation to the model: topography, specifically that of Yosemite National Park. The extreme changes in elevation – thousands of meters over a few kilometers – introduce steep gradients in the initial conditions of the model, which determine well-posedness, and they produce steep gradients in wind velocity, which determines stability, so the experiment tests the robustness of the algorithm. Of course, the actual results are not entirely realistic. They are mathematically realistic, but the model itself is not. Turbulence in valleys and solar radiation reflecting intensely from the quartz in the granite of the mountains are not factors that can justifiably be neglected in a weather model for Yosemite. Then again, one hardly needs a complex model to know that it will be windy atop the Half Dome. Yosemite is simply an interesting stress test for the model. And the model can be used as a stress test for other scenarios. That is the purpose of even the most sophisticated model: to provide an extreme scenario. It is an approximation of an approximation, but it guides engineering design. Consider the installation of solar panels. An ideal place for them is a desert. But the panels must mechanically withstand the maximum wind load of the desert, which is the force corresponding to the maximum wind speed. Because it assumes air is dry, the model of this study replicates desert conditions, and if solar radiation is added to the energy transport equation, it could certainly be used to predict the maximum wind speed. And if topography is added, it could facilitate the installation of solar panels on roofs of buildings in a city. The spatial step in such a model would have to be on the order of meters to preserve urban topography, so the largest stable time step of the Matsuno scheme with spatial finite-differencing would probably be impractically small. Fortunately, the Matsuno scheme is highly modular. It provides a framework for discretizing a property over time, but not space; the spatial discretization is completely arbitrary and must simply be evaluated twice with the proper arguments. The Fortran main program written for this study implements the Matsuno scheme in this manner. For each property at each time step, it calls the same external function twice, adjusting the arguments in accordance with the Matsuno scheme. The external functions could be rewritten to implement a more stable and accurate spatial scheme, and the main program would be unchanged, but its output would be a better approximation of the solution. The functions could be rewritten to include solar radiation and turbulence, and the output would be a better approximation of the weather.

2. The Program

Without turbulence, the equations of state and transport in a homogenous atmosphere are decoupled. Density appears in the energy and momentum equations only as an agent of diffusion, and without turbulence, there is no diffusion and thus no need to solve for density. The weather model is reduced to a system of the three transport equations governed by three properties: pressure, temperature, and wind velocity. The following discussion of their solution heavily references Mark Jacobson’s *Fundamentals of Atmospheric Modeling* (FAM), so it will begin by adopting the terminology of the text. Mass transport is called the continuity equation, energy transport is the thermodynamic energy equation, and momentum transport is the momentum equation. This study implements the finite difference discretizations of these equations provided in FAM to solve for the aforementioned three variables over a regional domain.

The domain itself is described by spherical coordinates in horizontal directions and sigma-pressure (σ-p) coordinates in the vertical direction. A regional patch on a sphere can be closely approximated by a simple plane, but spherical coordinates produce the Coriolis effect, which has a significant effect on wind currents at certain latitudes. The σ-p coordinates set the vertical 0 along any topography. Doing so makes the model more efficient and more flexible than if 0 were at sea level, or actually any constant elevation. The reason is the grid. Any discretization produces a grid of nodes at which system variables are computed, and a grid is represented in a computer by a rectangular array. For a grid of constant elevations, each position of the array along the vertical dimension corresponds to an elevation. Essentially, the array matches the grid. So imagine superimposing this rectangular array on an irregular surface. The array will inevitably pass over valleys and intersect mountains. To compensate for the valleys, the array must be expanded. To compensate for the mountains, the array must be filled with zeros at indices computed precisely to correspond to elevations that are underground. The result is an unnecessarily large array filled mostly with zeros and designed for a single topography. But for a grid of σ-p coordinates, each position of the array along the vertical dimension corresponds to a constant σ level. The first level at the ground is set to 1, the final level at the top of the model is set to 0, and all intermediate values are constants but follow the topography, any topography. The values do not have to be recomputed; it is the elevations that would change when a simulation is performed over a new region. So to recover temperature, for example, at a specific elevation, one can use the topography to compute the corresponding σ level and go to its position in the data array. A disadvantage is that while the σ levels are constant over the region, their elevations are not, so one cannot obtain an entire spatial profile of a variable at a particular elevation. But that is hardly a cost for the universality and compactness of the σ-p grid.

Numerically, the solution grid is a collection of nodes; between them is the void. Physically, however, the nodes are connected by edges that forms cells, and the cells are filled with air. And it is actually along the edges and in the centers of the cells that the properties of weather are estimated. Pressures and temperatures are always at cell centers. Horizontal east-west velocities are along the midpoints of longitudinal edges, while north-south velocities are along the midpoints of latitudinal edges. Finally, vertical velocities are at the centers of σ-surfaces. But numbers in a computer can only exist as nodes of a grid, so the model actually uses multiple overlying grids, each of the same cell size but each shifted to coincide with the solution point – a physical edge or center – of a property.

Boundary conditions must be set for each grid. Vertical wind velocity has a Dirichlet boundary condition; it must be 0 at the ground and the model top, and these are obviously the edges of an array corresponding to σ-p coordinates. The other variables, namely the horizontal velocities, temperature, and pressure, are governed by Neumann conditions, which specify the flux of mass, energy, and momentum across the boundary. Neumann conditions are implemented in finite difference schemes with virtual nodes that exist outside the model grid. For temperature and velocities, it suffices to set the values at these nodes set to the nearest internal node values from a previous time step. But to maintain the stability of the model, pressure at a virtual node must be followed *two* time steps back at the nearest internal node.

Boundary conditions specify the spatial problem, but the model is dynamic, and it requires initial conditions as well as a time-stepping scheme. This study assumes air is initially at rest and wind stirs only as temperature and pressure gradients evolve over time – in this study with the Matsuno scheme. This scheme is essentially a forward difference in time: if the time step is h and the quantity q_t is known at time t, then $q_{t+1} = q_t + hf(q_t)$, where f is the spatial discretization scheme. The Matsuno evaluates this equation twice, using the first result as the argument for computing the second the second. Formally, it computes an estimate $q_{est} = q_t + hf(q_t)$, and then marches in time based on the estimate: $q_{t+1} = q_t + hf(q_{est})$. Crucially, this second step of the Matsuno scheme propagates the original q_t, not the estimate. Thus an implementation of the scheme in a computer program must retain two copies of q. But both the estimation and the propagation steps can be computed with the same function f, making the scheme highly modular, an aspect that this study exploits.

With the grid, the transport equations, and their boundary conditions discretized, temperature, pressure, and wind can evolve over time, but the model does not output these exact properties. Horizontal wind is horizontal wind, but vertically it is modeled as a change in the dimensionless σ quantity over time, called $\dot{\sigma}$. But much like elevation can be computed from topography and the σ level, vertical velocity at any elevation can be computed from $\dot{\sigma}$, the σ

level, and the column pressure π_a of air. Column pressure is the output that replaces pressure. It is still measured in standard pressure units, but it describes an entire vertical column at any horizontal grid point. Of course, pressure at a specific elevation can be computed from it, once again from the σ levels. Such is the importance of the σ-p coordinate system. Finally, temperature is output as potential virtual temperature θ_v, which is the potential temperature that moist air would have if it were desiccated at constant density and pressure. This study models only dry air, so potential virtual temperature is equal to potential temperature, which in turn is equal to the temperature air would have at 1000 hPa, but in general the three measures of energy are distinct. Using θ_v in the equations is advantageous mostly because it eliminates various constants.

The model is implemented in Fortran90 as a main program that set initial parameters in subroutines then runs the Matsuno scheme, calling various external functions to estimate and propagate each variable. The files containing the source code are listed in the accompanying makefile, but the following subsections will explain them – their variables, interactions, and indexing schemes – in greater detail.

2.1. Variables

Global variables are available to all subroutines and functions – collectively called subprograms. The only variables that are passed to a subprogram as explicit arguments are those that are generic in the subprogram. The most notable example of a subprogram that uses generic arguments is one that implements the finite difference function $f(x)$ of a Matsuno scheme. The function itself is written in terms of x. But the main program calls it first with the previous value x_{prev} to obtain the estimate x_{est}, and later with x_{est} to obtain the current value x_{curr}.

Table 2.1 describes the global variables of the program and the local variables of the main function. The grid sizes give the number N of cells in each dimension, so the number of nodes will be N on a grid that coincides with centers (called midpoints in the code) and $N + 1$ on a grid that coincides with edges. Variables can have up to three indices corresponding to spatial dimensions. If the index is an integer of the form i, the variable must be evaluated on a cell center, and if the index is a fraction of the form $i + 1/2$, the variables must be evaluated on a cell edge. Centers and edges are treated as nodes on overlying but separate grids, so separate

Table 2.1: Primary variables

Symbol	Code	Meaning	Size
Global Variables (uvars, pvars)			
N_{lat}	`Nlat`	number of latitude rows (cell centers)	
N_{lon}	`Nlon`	number of longitude rows (cell centers)	
$\Delta\varphi$	`dlat`	increment in latitude	
$\Delta\lambda$	`dlon`	increment in longitude	
φ_0	`lat0`	latitude of SW corner	
λ_0	`lon0`	longitude of SW corner	
$\varphi_{i+1/2}$	`lat_edge`	latitudes of cell edges	$N_{lat}+1$
φ_i	`lat_mdpt`	latitudes of cell centers	N_{lat}
λ_i	`lon_mdpt`	longitudes of cell centers	N_{lon}
N_{vert}	`Nvert`	number of vertical levels (cell centers)	
σ	`sigma`	σ-levels	N_{vert}
N_t	`Nt`	number of time steps	
h	`h`	time increment	
p	`p_`	pressure	(dep. on suffix)
$P_{i+1/2}$	`PP_edge`	Exner pressure	$N_{lat}\times N_{lon}\times(N_{vert}+1)$
P_i	`PP_mdpt`		$N_{lat}\times N_{lon}\times N_{vert}$
F	`fluxF`	E-W flux	$N_{lat}\times(N_{lon}+1)\times N_{vert}$
G	`fluxG`	S-N flux	$(N_{lat}+1)\times N_{lon}\times N_{vert}$
Local Variables (weather.f90)			
π_a	`pi_a_`	column pressure	$N_{lat}\times N_{lon}$
θ_v	`theta_`	potential virtual temperature	$N_{lat}\times N_{lon}\times N_{vert}$
u	`uwind_`	E-W wind velocity	$N_{lat}\times(N_{lon}+1)\times N_{vert}$
v	`vwind_`	S-N wind velocity	$(N_{lat}+1)\times N_{lon}\times N_{vert}$
$\dot{\sigma}$	`sigmaDot`	vertical wind velocity	$N_{lat}\times N_{lon}\times(N_{vert}+1)$
Φ	`phi_`	geopotential	$N_{lat}\times N_{lon}\times N_{vert}$

variables are used for the same property on the different grids. To indicate that they represent the same property, such variables are named by attaching a descriptive suffix to a constant base. For example, the Exner pressure is stored in "PP_edge" at vertical edges, and in "PP_mdpt" at vertical centers (midpoints). The dimension to which the "edge" and "mdpt" suffix refers should be clear from the base variable and its size; Exner pressure is always evaluated at horizontal cell centers in the model, so its "edge" variable can only refer to vertical edges.

The naming convention of base and suffix to describe variations on the same property is used extensively for the local variables, so only their base names and sizes are tabulated. The suffixes mostly follow the Matsuno scheme described above. A suffix “prev” is the previous instance, “est” is the estimate, and “curr” is the new (current) instance. Other variations on a property might be “prev2” for boundary conditions that require a value from 2 backward time steps, and “interp” for interpolations.

Not tabulated are units. Within the Matsuno scheme, properties are always given in SI units without prefixes, and longitudes and latitude are given in radians. The only caveat is the appearance of pressure in hectopascals in the Exner function. Indeed hectopascals may be more tangible units for pressure, as degrees might be for coordinates, but keeping the quantities in base form for computations avoids needless scaling of the transport equations. Before the intensive iterations begin, though, the user interface of the model does incorporate the more common units.

2.2. Program Structure

The program consists of multiple subroutines, functions, and modules. This section describes the purpose of all but the “calc_X.f90” files, which are used by the Matsuno scheme to propagate property X. They are introduced as generic functions in this section, but their nuances are discussed in 2.3.

2.2.1. User Interface

While the program lacks a proper, precompiled interface, where one could enter various parameters in any units, the “uvars” module contains all customizable *user* variables. In addition to the grid and time-stepping parameters, the user must choose a uniform pressure for the top of the model and then manually find and enter empirical data for an elevation level just below the elevation where the model top pressure occurs. Elevation here is entered here in kilometers, coordinates in degrees, and pressure in hectopascals. The module also allows the user to incorporate topography into the model. If “topo” is left as 0, the terrain stays flat. Any other value requires the user to provide a the name of a text file containing a digital elevation model (DEM) of the model domain. The DEM must abide by a strict format. The elevations must be in meters. They must be discretized over the same grid as pressure, that is, the elevations must be given at the centers of the model grid. And the DEM file must be “upside-down;” the SW corner must be in the (1,1) position of the data matrix, and the SE corner must be in the (1,N) position.

The program does not contain a routine to catch any errors; the user is responsible for the consistency of the DEM.

Finally, the module contains a block of variables that tell the program what outputs to print to a file and how to name the file. The printing operation is entirely contained in the "printToFile" subroutine, and it should be changed along with the corresponding variables in "uvars" as the model is repurposed. Currently, it is designed to print the properties of weather at 8 specific times, given in the "printTimes" array, and 3 σ levels, given in the "printLevels" array. The variable "hindex" was used to distinguish executions of the program with different time steps.

The initial pressure distribution in the model region should be made more open to the user, as the original Gaussian peak is hardly representative of real weather, but it has been left in the "initializePsurf" subroutine. For this study, the central Gaussian hill of pressure was in fact changed to a longitudinal pressure front, as described in Section 2.2.3., and a user could follow that procedure to construct any initial distribution.

2.2.2. Global Variables

The user variables are global, but they are also a subset of the problem variables, or global variables contained in the "pvars" module. This module USES the "uvars," so all subroutines need only USE "pvars" to gain access to all the global variables, but the two are separated because the problem variables should not be modified by a user. They specify the physical constants used throughout the model, and they declare the model property arrays in sizes consistent with their grids. The properties that are kept global are those that are not explicitly governed by the transport equations, specifically the fluxes F and G, and the Exner pressure P. Both are evaluated twice during each time step, but they are *entirely* reevaluated based only the most recent weather properties. In contrast, those weather properties, the ones driven by transport, are effectively *adjusted* from their value at a previous time step.

2.2.3. Initialization

The main program is stored in "weather.f90." It begins by declaring the local variables described in Table 2.1 – in all their many spatial and temporal flavors – as well as a block of pressure and Exner pressure variables; these are used only to initialize the model, as will shortly become clear.

The first call is to the subroutine “readTableB1.” As the name implies, this routine simply reads a digital copy of Table B.1 from FAM into the set of problem variables with suffix “dat.” These variables now contain empirical data describing the atmosphere over elevation increments small enough that one can interpolate the data over them with reasonable accuracy. Such interpolation facilitates the computations of the σ values. The final values are given by Equation 7.4, but they depend on pressures corresponding to various elevations. In a hydrostatic atmosphere, such as the one initially at rest in this model, pressure changes with elevation in accordance with Equation 7.3, which the program evaluates with values of density and gravitational linear interpolated from the empirical data. Linear interpolation is not an intrinsic function in Fortran90; it is performed by the custom subroutine “interp1D,” which takes data vectors x and y as arguments and uses them to predict y_i corresponding to a given x_i. The interpolation algorithm depends on the two points in x closest to x_i, which are found by another custom subroutine, “find2els.”

The σ values complete the vertical grid; the next step is to construct the horizon one. All input coordinates are here converted to radians for the remainder of the program. The midpoint grids are constructed by incrementally adding $\Delta\lambda$ or $\Delta\varphi$ to λ_0 and φ_0, respectively. Explicit coordinates for the edge grid are only required in latitude, so the $\varphi_{i+1/2}$ points are constructed by the same incremental process but beginning with $\varphi_0 - \Delta\varphi/2$.

Topography is set by a condition triggered by a user variable. If the variable is set to 0, there is no topography, and the surface geopotential is set to 0. Otherwise, elevation data is read from an input DEM. Fortran90 conveniently reads a file containing a two-dimensional array into a variable containing a two-dimensional array with a single line, but it does so *column-wise*. That is, the first row of the file becomes the first column of the array. The program therefore creates a dummy variable $\Phi_{surf,temp}$ of size $Nlon \times Nlat$ to store the result of the read, and then it transposes the dummy into the Φ_{surf} variable of the actual grid size, $Nlat \times Nlon$. At this point, Φ_{surf} is not geopotential, the measure of topography that is actually used in the model; it is just the elevation. Before converting it to geopotential, the program uses this raw elevation data to adjust the “base” pressure of the model. This base pressure is set to 1000 hPa at sea level, and it is later perturbed in the initial condition, but if there is topography, then the base is no longer uniform. It must be adjusted for the natural decrease of pressure with elevation, which is approximately 100 hPa per kilometer. Such a rough approximation suffices to establish a base pressure that will further be perturbed to create the initial condition.

Indeed the initial condition should be realistic, but it need not be accurate; this study is interested in the *model* being accurate given *any* initial condition. The initialization is performed in the “initializePsurf” subroutine. Despite pressure being a three-dimensional property, the initial distribution is computed in a loop only over the surface. That is because the model proper does not use absolute pressure; it uses the *column* pressure, which is a two-dimensional property computed as the difference between absolute pressures at the ground and the top of the model. In σ-p coordinates, this flat property fully describes pressure throughout space.

Temperature is initialized based on the initial pressure. The algorithm consists of the following steps.

- Compute pressure “p_edge” at vertical boundaries over the entire spatial domain.
- Loop over all vertical cells to compute pressure “p_mdpt” at cell centers:
  - at each σ level, convert edge pressure “p_edge” to Exner pressure “PP_edge” with Equation (7.10);
  - use the two vertical edge Exner pressures “PP_edge” and absolute pressures “p_edge” of a cell to estimate central Exner pressure “PP_mdpt” in that cell with Equation (7.11);
  - convert central Exner pressure to central pressure with an inversion of Equation (7.10).
- Interpolate empirical temperature data over pressure to obtain temperature corresponding the pressure at the center of each vertical σ level, then convert temperature to potential virtual temperature with Equation (2.99).

Thus the interpolation subroutines have more than one application in this program. The above procedure for initializing temperature differs slightly from that suggested by FAM. The text requires one to interpolate the empirical data to find temperature only in a corner vertical column of the model, and then to assume that temperature in every other column is equal to that in the corner. This method avoids interpolating in every column at the cost of a less accurate temperature profile. Then again, the initial condition must simply pose a gradient to activate the model; accuracy matters only in the results.

Two variables remain unaccounted for: pressure and geopotential, two time steps back. Those quantities is unknown, but their purpose in the Matsuno scheme is to stabilize pressure at the boundary in the momentum equation. Replacing them with their values from one step back for a single iteration should hardly destabilize the entire model. Therefore the pre-initial pressure is set equal to the initial pressure for the first time step; the pre-initial geopotential is set with an "if" statement in the main time loop for the pedantic reason of avoiding one superfluous version of its outside the loop. Then, after that first time step, the quantities two steps back can certainly be stored and stability restored.

2.2.4. Matsuno Scheme

Discretized in space, the equation governing the transport of a conserved quantity q assumes the form $\partial q/\partial t = f(q)$, where f is the spatial scheme. A forward-difference approximation, accurate of order 1, simply discretizes the definition of the derivative and rearranges terms to solve for a new value of the quantity: $q_{t+t} = q_t + hf(q_t)$. This method is unstable, so the Matsuno scheme separates it into two steps: estimation, $q_{est} = q_t + hf(q_t)$; and correction $q_{t+1} = q_t + hf(q_{est})$. The program written for this study abstracts these formulations further into $q_{est} = \mathcal{F}(q_t, q_t)$ and $q_{t+1} = \mathcal{F}(q_t, q_{est})$. The functions $\mathcal{F}$ are encoded in Fortran90 as external functions that accept generic arguments, and the main program is responsible for passing to them the correct instances of the local variables. These variables all follow the same naming convention: the base property, followed by a suffix: "prev" corresponds to q_t, "est" to q_{est}, and "curr" to q_{t+1}. The main program must also reevaluate the global variables with the appropriate arguments. As discussed in Section 2.2.2., only the properties that explicitly govern the transport equations – pressure, temperature, and wind speed – are propagated through the Matsuno scheme. The auxiliary fluxes F and G, and the Exener pressures P are reevaluated entirely and independently of their previous states at each Matsuno step. They do not need suffixes, as they appear in the functions $\mathcal{F}$ with their single, instantaneous values.

The ordering of the function calls is evident from the code. The workflow of both the estimation and correction consists of the following: fluxes → pressure → vertical velocity → Exener pressure → temperature → geopotential → wind velocity. The ordering of arguments to the functions seems more complicated, as the call to the u calculator takes 11 inputs, but this call can still be written in the general Matsuno form $q_{t+1} = \mathcal{F}(q_t, q_{est})$ if the arguments are treated as vectors of arguments corresponding to the appropriate version of the system properties. In the correction call to the u calculator, q_t is just u_t and all other variables are estimated, but in the correction call to the θ calculator, q_t is both $\pi_{a,t}$ and θ_t because the energy equation is solved for θ but propagates $\pi_a\theta$ as a single quantity. Such inner machinations of the property calculators are discussed in greater detail in the following section.

2.3. Property Calculators

Subroutines and functions in Fortran are identical in almost every way. They can accept generic arguments, they can declare local variables that will be lost when they end, and they can use global variables from modules that will be saved. They serve the same fundamental purpose: to divide a program into manageable, modular tasks. The main, if the not the *only*, difference is that a function returns a value while a subroutine does not. And that is why functions are the ideal construct for temporal finite difference calculators. The Matsuno scheme involves at least three versions of the properties it propagates: a previous value, an estimate, and the next value. A function call clearly shows how the previous value is used to obtain the estimate, and how both are then used to update the next time value. A subroutine could certainly be made to work, but the variable to be updated would have to be stored globally and juggled around after every call. It is much more suitable for updating the flux and Exener pressure. These properties do not propagate in time. They are completely reevaluated at each time and each Matsuno step, and by their designation as global variables, their instantaneous values are made available to all other functions. A subroutine is optimal for updating global variables because they need not be returned; they are simply stored in the global workspace.

This section discusses the equations and indexing schemes that are implemented in the subprograms called to calculate properties during each Matsuno step. The source codes are in the files named “calc_X.f90,” with X being the property name. Additionally, the module “funcs” provides the interface for all of the functions. The function interface in Fortran90 is analogous to the function declaration in C; each function is opened with generic arguments and contains a body that specifies the type and size of the arguments and the return value. Most of the arguments of the property calculator functions are multi-dimensional arrays, so their sizes are left arbitrary in the interface by use of the colon. The outputs are also arrays, but their sizes often coincide with the sizes of the inputs, so these can be preallocated. The only except is the $\dot{\sigma}$

calculator. Its only input is π, which is two-dimensional, so the length of the third dimension of $\dot{\sigma}$ is unknown. An easy workaround is to pass the number of vertical layers in the model as a dummy argument. It works, but it triggers a warning from the compiler because it is unused in the function proper. Ignore this warning; the 8 bytes of a wasted integer are nothing relative to the scale of the problem.

2.3.1. Flux

The subroutines "fluxF" and "fluxG" use the most recent values of pressure and wind velocity to solve Equations (7.15) and (7.16) for F and G, respectively. Boundary values are computed with a conditional that implements Equation (7.17) and (7.18) on the northern and western edges and analogous forms on the southern and eastern edges. Of note in these subroutines is that the pressure and flux grids do not perfectly coincide. Consider, for example, the longitudinal index i. When $i = 1$, $F_{i+1/2}$ is in the second column of the F array, but $\pi_{a,1}$ is just the first column of the π_a array. So the term $\left(\pi_{a,i} - \pi_{a,i+1}\right)$ must be indexed in code as `(pi_a(i-1) – pi_a(i))`. This difference in indexes between the formulation of the equations in the texts and their implementation in code recurs in the property calculators whenever grids do not overlap.

2.3.2. Column Pressure

In both the estimation and correction steps of the Matsuno scheme, the pressure calculator "pi_a_calc" is called with "pi_a_prev" – the old value – as its only argument. That is because the estimated value is already incorporated in the fluxes F and G during the correction step. Of course, the fluxes do not coincide spatially with the pressure, so the indices differ. Unlike in the flux calculator, though, this time the indices are determined by the values they assume under π_a. When $i = 1$, $\pi_{a,i}$ is in the first column of the π_a array, but $F_{i+1/2}$ is the second. So the term $\left(F_{i+1/2} - F_{i-1/2}\right)$ in Equation (7.14) becomes `(F(i+1) – F(i))` in the code.

2.3.3. Vertical Velocity

Vertical velocity is given by Equation (7.21), and it is implemented as such, but its governing equation actually describes $\dot{\sigma}\pi_a$. Thus the call to "sigmaDot_calc" during the Matsuno correction step must retain “pi_a_prev” as the first argument but use “pi_a_est” as the current value driven by flux.

2.3.4. Exener Pressure

The Exener pressure calculator “PPcalc” is called just before the temperature calcaultor, partly because that allows it to use the most recent pressure values as arguments, but also because it is fundamentally connected to temperature. Indeed the subroutine replicates the algorithm described in Section 2.2.3 for initializing temperature. Absolute pressures “p_edge” are computed at vertical boundaries over the entire spatial domain using the definition of column pressure. A loop over the vertical centers is then run in which Equation (7.10) is used to compute Exener pressures at vertical edges and Equation (7.11) is used to interpolate these values for the Exener pressure at the midpoint between the edges. The final step in the original algorithm – the recovery of pressure at the midpoints – is omitted because in this case it the Exener pressure is the desired output. But being in a subroutine, it is not returned; it is reevaluated and saved to the global workspace.

2.3.5. Temperature

Potential virtual temperature is given explicitly by Equation (7.27), but much like $\dot{\sigma}$, it was not the quantity transported by the energy equation; that quantity was $\pi_a \theta_v$, and so the “previous” values of π_a must be used in the leading term of the discretization during the Matsuno correction step. “Estimated” values can be used in the other terms. Furthermore, Equation (7.27) requires values of θ_v at vertical edges as arguments. These are obtained by calling the interpolator “theta_interp_calc” before “theta_calc” in the main program and storing the output in an array designed to hold vertical edges. The interpolating function accompanies “theta_calc” in the source file and implements Equation (7.11).

2.3.6. Geopotential

Geopotential is always derived from the most recent temperature, and it is not actually a transported quantity, but it is evaluated by a function, “phi_calc,” rather than a subroutine because two instances must always be available for use in the wind velocity equations. Those equations only require its values at vertical midpoints, but Equations (7.61) through (7.63) dictate that the geopotential must be solved at both midpoints and edges.

2.3.7. Wind Velocity

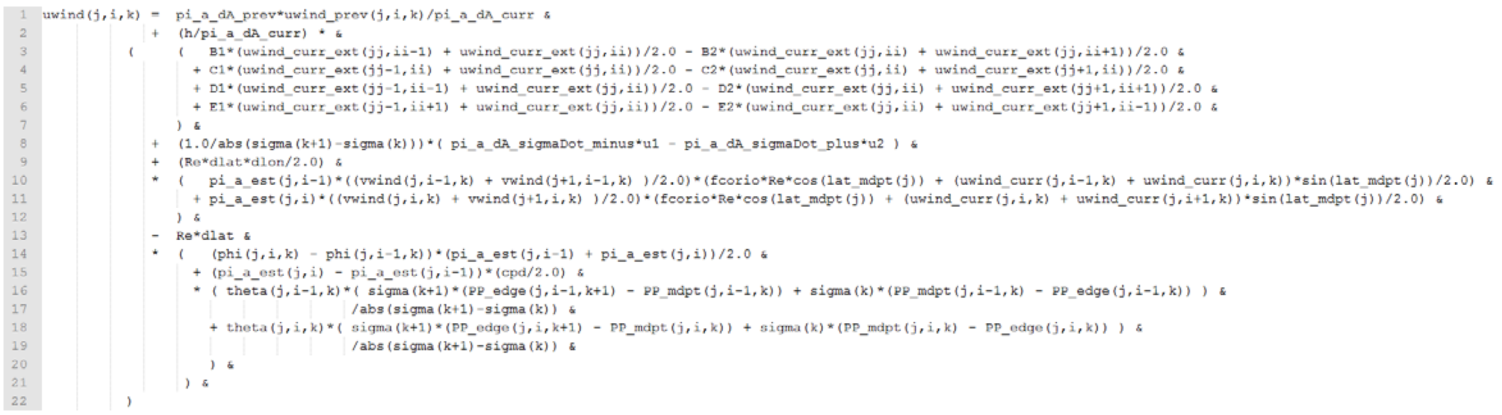
The latitudinal component of horizontal wind velocity – that is, the one directed from east to west – is computed by the code excerpted in Figure 2.1. In the following explication of this code, variables are consistent with the namespace of the calculator function, not the main function. For example, they both use variations of column pressure. The calculator uses the names to indicate the roles of the variations: the "previous" and "current" versions of column pressure are used only in the time difference term, while the "estimated" version is used only in the spatial discretization terms. The main functions uses the names to indicate the instance of the column pressure, and it assigns the instances to the arguments of the calculator consistently with the Matsuno scheme and the order of arguments. In the calculator namespace, the order is:

```
uwind_calc(pi_a_prev,  pi_a_prev2,  pi_a_curr,   pi_a_est).
```

In the estimate step of the Matsuno scheme, the main function calls the calculator with:

```
uwind_est = uwind_calc(pi_a_prev,  pi_a_prev2,  pi_a_est,   pi_a_est).
```

At this point, the corrected, "current" column pressure has not yet been evaluated, and it is the estimate which is used in its place in the calculator. During the correction step, that argument does become the most recent pressure. The pressure from two time steps back is also updated; the "est" is effectively one step back, so the "prev" value in the main function is used for the "prev2" value in the calculator. But the "prev" value in the calculator remains the "prev" value from the main function because, to emphasize, it is only used in the time step of the Matsuno scheme, which does not involve estimation.



```
uwind(j,i,k) =  pi_a_dA_prev*uwind_prev(j,i,k)/pi_a_dA_curr &
              +  (h/pi_a_dA_curr) * &
        (        (   B1*(uwind_curr_ext(jj,ii-1) + uwind_curr_ext(jj,ii))/2.0 - B2*(uwind_curr_ext(jj,ii) + uwind_curr_ext(jj,ii+1))/2.0 &
                   + C1*(uwind_curr_ext(jj-1,ii) + uwind_curr_ext(jj,ii))/2.0 - C2*(uwind_curr_ext(jj,ii) + uwind_curr_ext(jj+1,ii))/2.0 &
                   + D1*(uwind_curr_ext(jj-1,ii-1) + uwind_curr_ext(jj,ii))/2.0 - D2*(uwind_curr_ext(jj,ii) + uwind_curr_ext(jj+1,ii+1))/2.0 &
                   + E1*(uwind_curr_ext(jj-1,ii+1) + uwind_curr_ext(jj,ii))/2.0 - E2*(uwind_curr_ext(jj,ii) + uwind_curr_ext(jj+1,ii-1))/2.0 &
                 ) &
              +  (1.0/abs(sigma(k+1)-sigma(k)))*( pi_a_dA_sigmaDot_minus*u1 - pi_a_dA_sigmaDot_plus*u2 ) &
              +  (Re*dlat*dlon/2.0) &
              *  (   pi_a_est(j,i-1)*((vwind(j,i-1,k) + vwind(j+1,i-1,k) )/2.0)*(fcorio*Re*cos(lat_mdpt(j)) + (uwind_curr(j,i-1,k) + uwind_curr(j,i,k))*sin(lat_mdpt(j))/2.0) &
                   + pi_a_est(j,i)*((vwind(j,i,k) + vwind(j+1,i,k) )/2.0)*(fcorio*Re*cos(lat_mdpt(j)) + (uwind_curr(j,i,k) + uwind_curr(j,i+1,k))*sin(lat_mdpt(j))/2.0) &
                 ) &
              -  Re*dlat &
              *  (    (phi(j,i,k) - phi(j,i-1,k))*(pi_a_est(j,i-1) + pi_a_est(j,i))/2.0 &
                   + (pi_a_est(j,i) - pi_a_est(j,i-1))*(cpd/2.0) &
                   * ( theta(j,i-1,k)*( sigma(k+1)*(PP_edge(j,i-1,k+1) - PP_mdpt(j,i-1,k)) + sigma(k)*(PP_mdpt(j,i-1,k) - PP_edge(j,i-1,k)) ) &
                                          /abs(sigma(k+1)-sigma(k)) &
                     + theta(j,i,k)*( sigma(k+1)*(PP_edge(j,i,k+1) - PP_mdpt(j,i,k)) + sigma(k)*(PP_mdpt(j,i,k) - PP_edge(j,i,k)) ) &
                                          /abs(sigma(k+1)-sigma(k)) &
                     ) &
                  ) &
        )
```

Figure 2.1

Line 1 of the code in Figure 2.1 is the time step; it corresponds to Equation (7.32) in FAM. Just like vertical velocities and temperatures, the horizontal velocity is actually discretized in time as one factor in a product with column pressure and grid cell area. But unlike vertical velocity and temperature, horizontal velocity does not exist on the same solution grid as pressure. Therefore, the aforementioned products are interpolated by the method given by Equations (7.38) and (7.39). This method requires values of pressure at six adjacent nodes on its solution grid, and thus a new problem arises from the disconnect between the grids: the velocity might be on an interior node while Equation (7.38) calls for external values of pressure. Of course, this dilemma could be solved by copying the code of Figure 2.1 into nested conditionals that check whether all the pressure nodes are available, but given the length of the code that is – inadvisable. Instead, the calculator stores the product of pressure and area in a local variable that it evaluates in its own nested conditionals, one set for each term in the interpolation. The boundary problem ultimately assume that pressure at an external node is equal to pressure at the nearest internal node. Doing so is equivalent to assigning new weights to the terms in the interpolation, so when the external nodes are assigned internal values, they should be treated as internal nodes. The distinction is subtle but important because it implies that Equation (7.39), which gives the area of a grid cell, should be incorporated into the interpolation, with the coordinates of the cell adjusted along with the nodes.

Lines 3-6 of the velocity equation represent advection. They involve the fluxes F and G as well as the "est" value of u at multiple nodes adjacent to the one at which u is being updated. And, once again, the nodes do not coincide; F and u exist on a different solution grid from G. So the equations governing advection – (7.33) and (7.41-7.44) – could be evaluated in nested conditionals. But the conditionals would have to be separate from those used for pressure because pressure exists on its own, third solution grid. And the aforementioned governing equations are much longer than the simple interpolations for pressure. So to solve this boundary problem, the calculator simply expands the grid so that boundary nodes become internal nodes. Arrays of u, F, and G at each vertical σ level are passed to the external function "extendMat," which increases the size of the arrays by 2 in each direction and sets the new boundary values equal to the old boundary values. These extended arrays are then accessed with indices $ii = i + 1$ and $jj = j + 1$. Thus if F_{ext} is the extended version of F, a request for $F_{i,j}$ is equivalent to $F_{ext,(ii,jj)}$. And this is useful because accessing F at $i = 0$ is impossible, while accessing F_{ext} at

$ii = 0$ is not. Thus the code in Figure 2.1 implements advection with the same indexing scheme as Equation (7.33) and avoids any conditionals and issues of disconnected grids on problem boundaries.

Line 8 is the transport of momentum, Equation (7.34). It requires the interpolation given in Equation (7.40) of the product of pressure, area, and vertical velocity. Fortunately, the calculator has already handled the product of pressure and area, so it can reuse those boundary conditions in the new interpolation.

Lines 9-12 express the Coriolis effect, and lines 13-19 complete the statement of Newton's Second Law by adding the external force that drives momentum: pressure. The variables in these code blocks are not all on the same solution grid as wind speed, but the indices used to access the wind array never force the other variables off of their grids, so the nested conditionals and extended arrays from before are not necessary. This time, the entire code block of Figure 2.1 is repeated in a single conditional that handles the pressure gradient at the boundary. When u must be evaluated where pressure is unavailable, the pressure gradient term is replaced with Equation (7.46); lines 13-19 are simply cut out and replaced with the new formula, which sets external values to internal values from two time steps back; lines 9-12 are adjusted in the typical fashion of setting external values to nearest internal values.

2.4. Efficiency

The program is designed to be readable, but efficiency is maximized where possible. Equations are written so that the code resembles the text. In the text, the indexing scheme (i, j, k) considers i as the x-coordinate of an array, j as the y-coordinate, and k as the z-coordinate. In computer code, i would actually be the index of the row of an array, that is, the y-coordinate. So to be consistent with the text, the program used in the study is written to access arrays in the form (j, i, k). Loops are therefore written in the order DO k … DO i … DO j rather than the more common k, j, i. This ordering maximizes use of the cache line. Other sources of efficiency come from storing repeated constants in local variables. The area of a grid cell changes in the wind velocity calculator with latitude, but the product of Earth's radius and the grid size can be precomputed to avoid repeating a constant operation within the loop. Function calls are generally expensive because they involve jumps across the stack, but they avoid the use of global variables

– which are not only expensive in their way, but would also require iterative reassignment of pointers.

III. Error Analysis

First test the program with flat terrain. Set the grid to match that of Yosemite, so that the test can predict whether the model will be stable for the practical discretization. $N_{vert} = 15$, $N_{lon} = 40$, and $N_{lat} = 40$ with $\Delta\varphi = \Delta\lambda = 0.025°$ and $\varphi_0 = 37°$, $\lambda_0 = -120°$. Set $h = 0.5$ seconds. Wind speeds are low here, but the discretization is very fine, and we want the test to be comparable to the Yosemite simulation, which will have much higher wind speeds. Initialize pressure with the Gaussian curve given in Problem 7.12. Results are shown in Figures 3.1 through 3.3. Arrows in the velocity vector plots are normalized so that a unit length on the figures corresponds to 0.30 m/s. This is about the maximum reached at the 7th σ level. Wind speeds are much higher at both the model top and ground level, where they reach about 1 m/s. Temperature results are given in K and column pressures in hPa.

The model has realistic results, but are they accurate? Are they consistent with the mathematical theory? To test, run three sets of simulations over the previous grid: one at $h = 0.5$, one at $h = 1$, and one at $h = 2$ seconds. Record u, v, θ_v, and π_a at Matsuno iterations corresponding to the same real time. For example, if we record the $h = 0.5$ at iteration 100, then we must record $h = 1$ at iteration 50 and $h = 2$ at iteration 25. Treat the $h = 0.5$ results as the best approximation of the true solution of the model; call them q_0, with q being each of the output properties. Call the $h = 1$ results q_1 and the $h = 2$ results q_2. These results are 3D arrays. For the k^{th} σ level, compute the L2 errors $e_{1,k} = \|q_0 - q_1\|$ and $e_{2,k} = \|q_0 - q_2\|$. Average them over the k-dimension to get average L2 errors e_1 and e_2. The Matsuno scheme is first order accurate in time. So scaling h by a factor of x should scale e by x as well. In our case, this means that the relative error e_2/e_1 should be about 0.5, since we scaled h by 0.5 between the simulations. The actual value of the relative error over time is shown in Figure 3.4. It is somewhat lower than 0.5 because the numerical solution at $h = 0.5$ is certainly not the true solution, but it is close. Furthermore, it is stable over time; in fact, it decreases slightly for v between 1 and 2 hours. So we can claim our program is consistent with the theory of the Matsuno scheme. In other words, the program is stable and accurate.

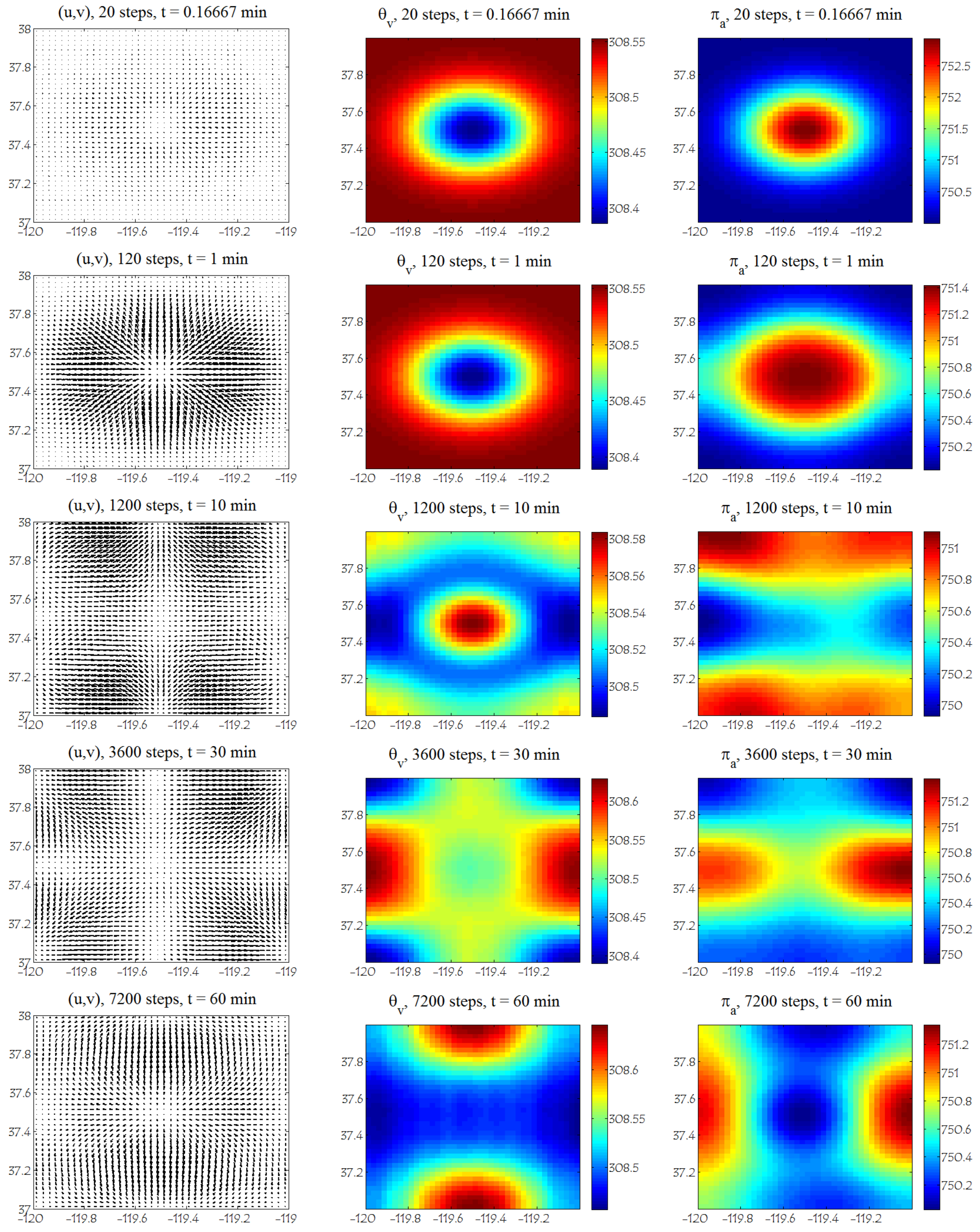


Figure 3.1: Flat terrain, 7th σ level

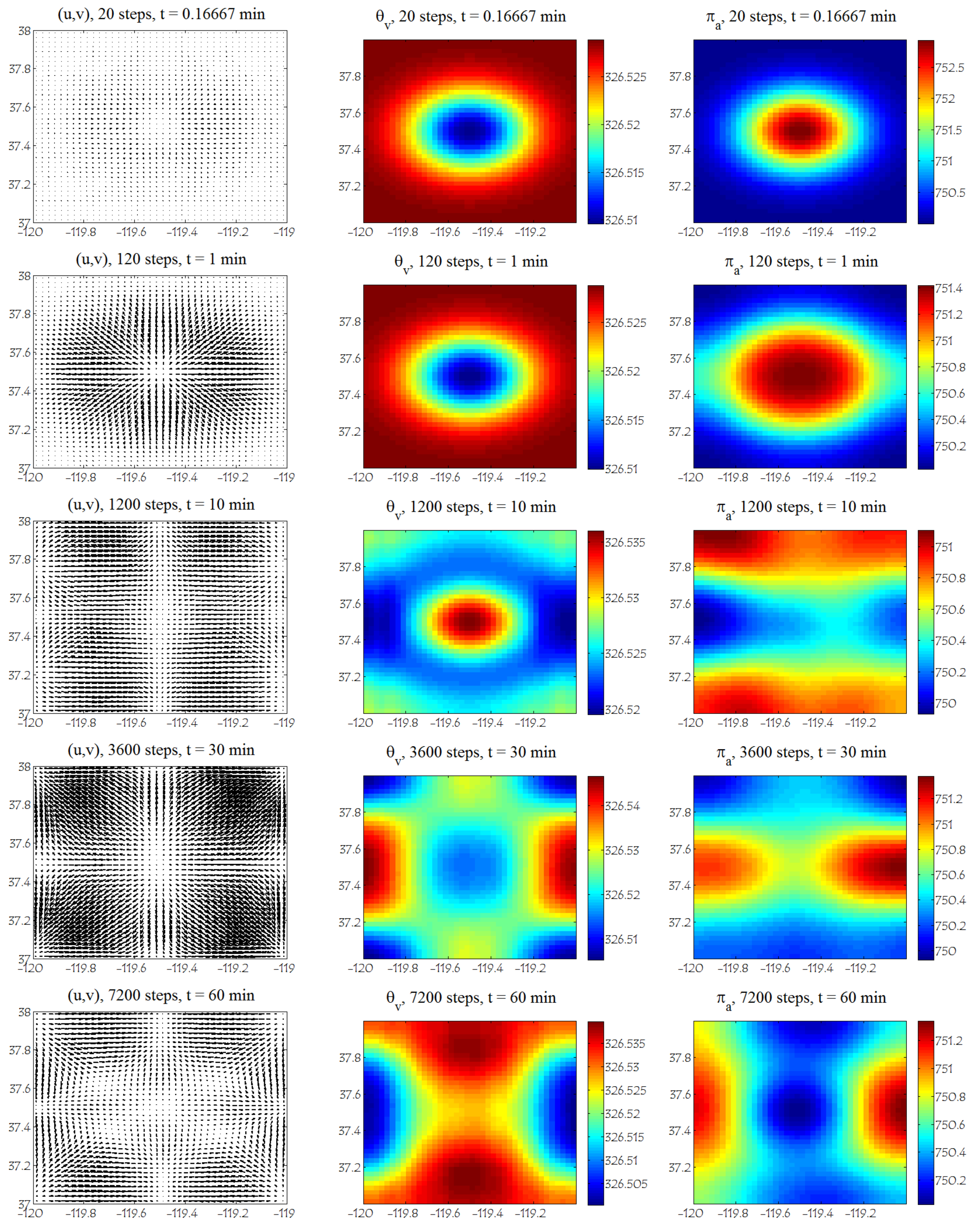


Figure 3.2: Flat terrain, 1st σ level (model top)

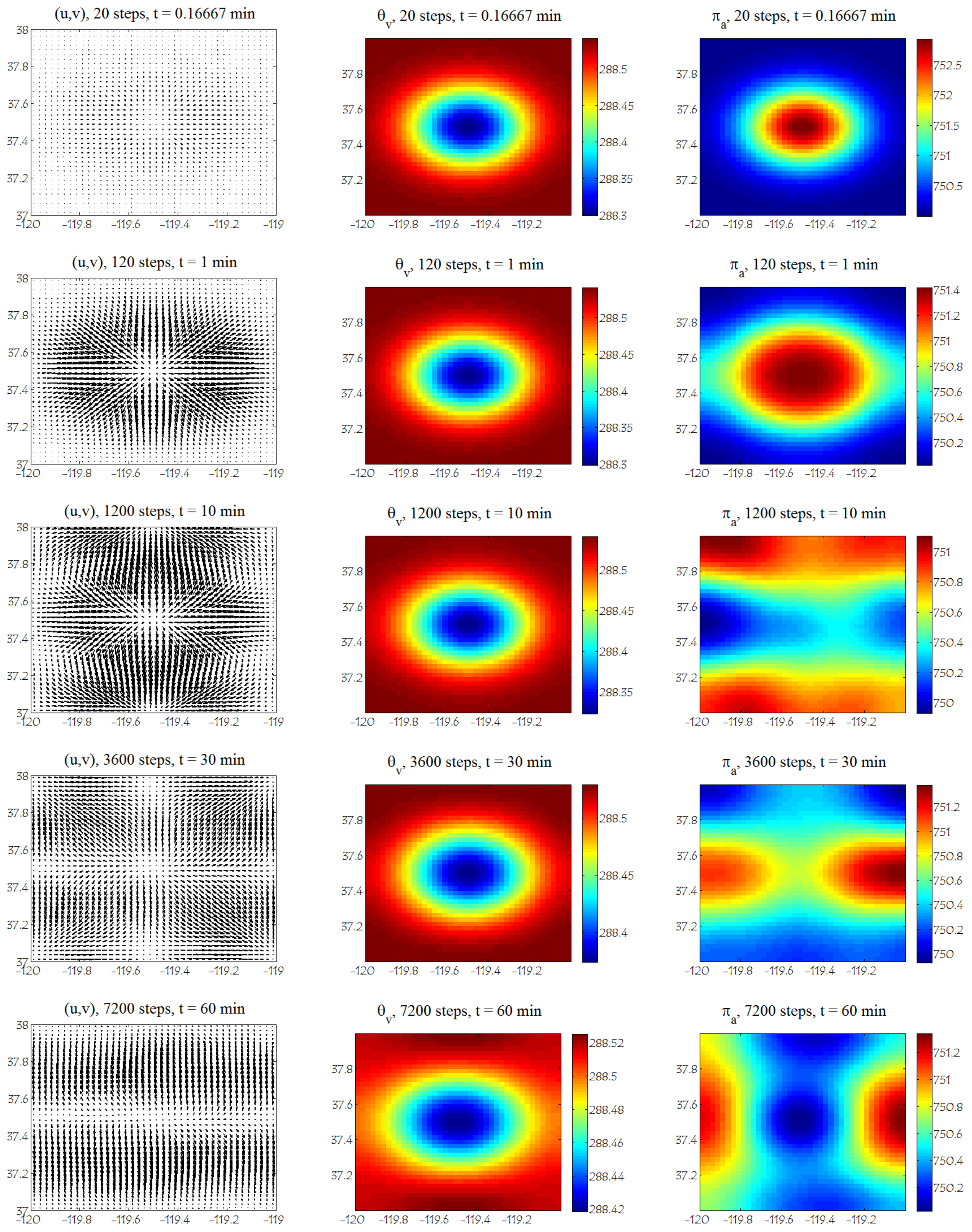


Figure 3.3: Flat terrain, last σ level (ground)

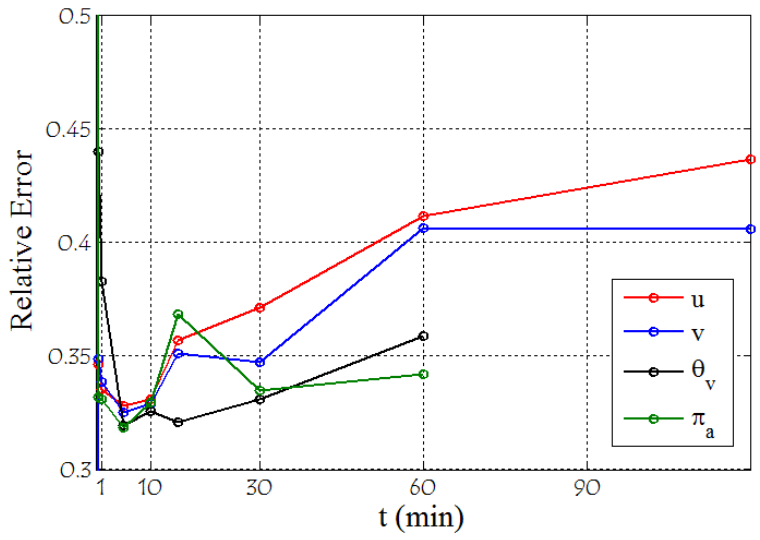


Figure 3.4

IV. Application: Yosemite

I obtained a DEM covering Yosemite from the USGS; it is the Mariposa-west quadrant. The DEM gives elevations every 3 arc-seconds, or 1/1200 of a degree. This would create an intractably large grid, so I resampled the DEM to the previous model grid. Figures 3.5 through 3.7 show results using the same parameters as in the test case. Velocity arrows are normalized so that the maximum speed at the 7th σ level is 25 m/s. At other levels it reaches 50 m/s, which is certainly reasonable for the mountainous terrain. This first simulation of Yosemite was run using an initial pressure gradient given by the Gaussian hill superimposed over base pressure, which was adjusted from the 1000 hPa at sea level according to the topography, with a 100 hPa decrease for every 1 km. I created another initial pressure gradient to use in a second simulation. It is shown in Figure 3.8. It is designed to simulate the decrease in pressure that occurs on mountains in the morning, when the mountains are warmed faster than the valleys. However, the results of a simulation with this gradient are hardly any different from those of the original simulation, and they are therefore not shown here. The reason is that the topography dominates the initial pressure gradient. A Gaussian bump or ridge whose magnitude is 3 hPa almost vanishes almost immediately. Future simulations should increase the magnitude of any initial pressure perturbations.

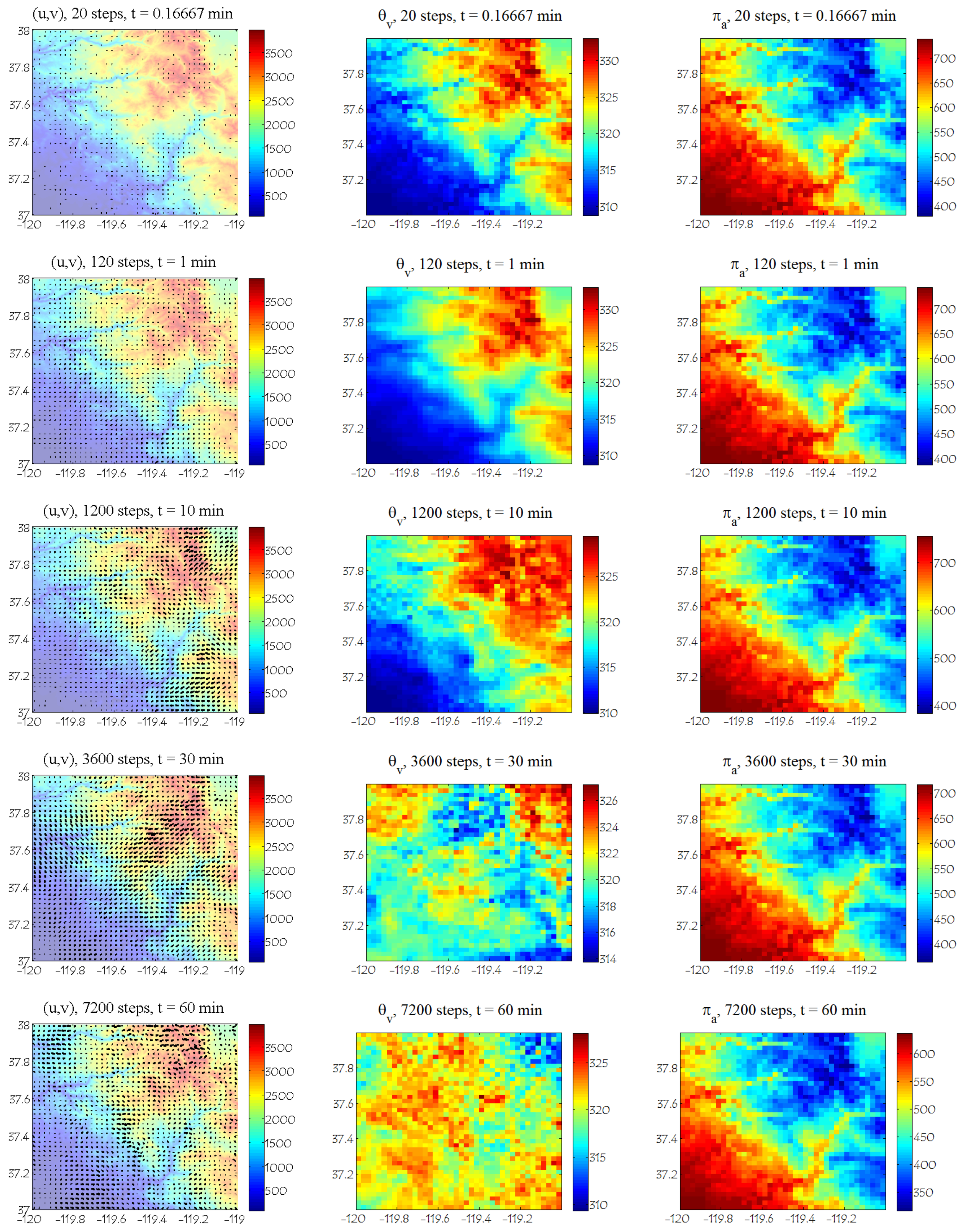


Figure 3.5: Yosemite, 7th σ level

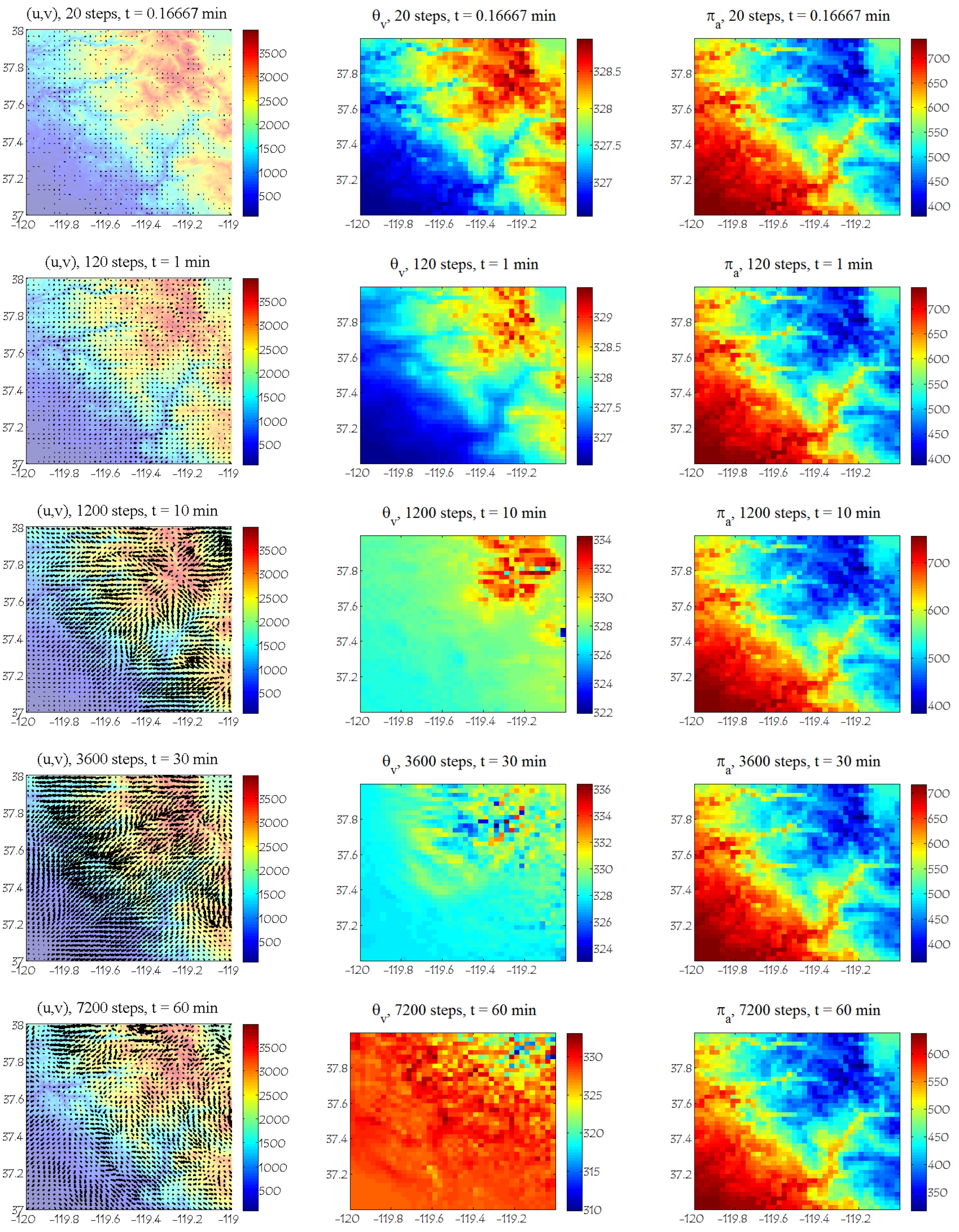


Figure 3.6: Yosemite, 1st σ level (model top)

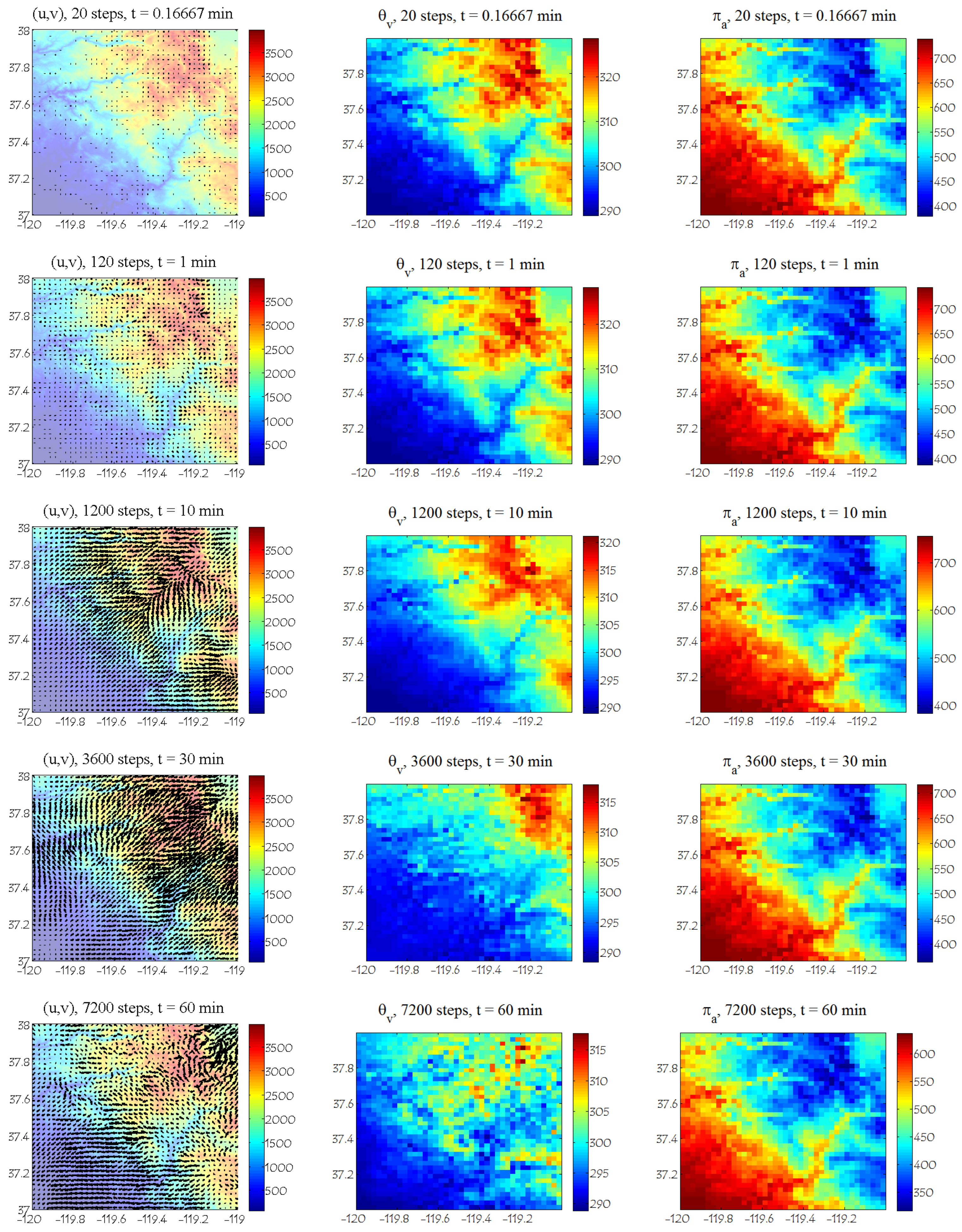


Figure 3.6: Yosemite, last σ level (ground)

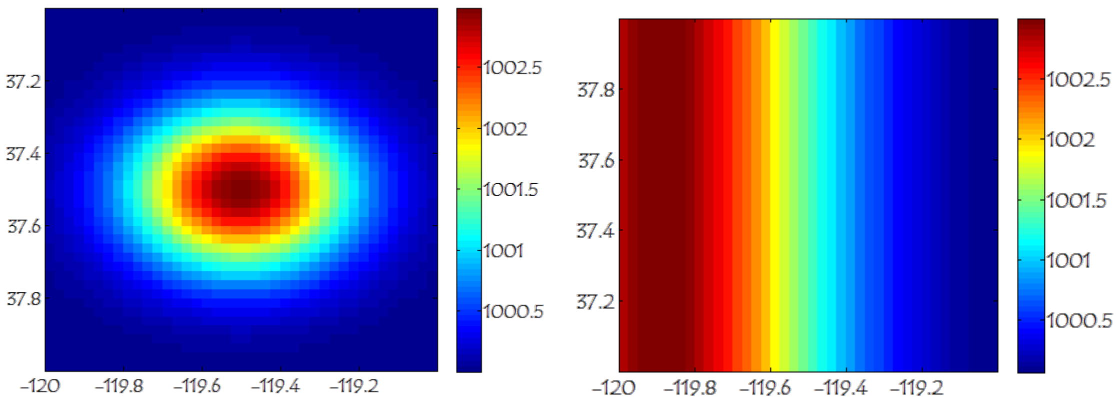


Figure 3.8: Initial pressure perturbations from a base of 1000 hPa at sea level. On the left is the Gaussian bump used in the original test case and the first Yosemite simulation. On the right is the ridge used for the second Yosemite simulation. Because the base level is adjusted from 1000 hPa in accordance with topography, the perturbation was almost immediately absorbed by the prevailing dynamics of the model.