Lecture Notes for

Production Quality Computational Multi-Physics Code Development (DRAFT)

Scott Runnels

August 7, 2018

Acknowledgments

I sincerely thank Los Alamos National Laboratory (LANL) for funding the development of these lecture notes and for providing funding to allow me to teach the class. In particular, I thank the leadersship in LANL's Computational Physics division (XCP) and its supervising organizations (ADX, PADWP), including Bob Webster, Michael Bernardin, Mark Chadwick, Bob Little, Ed Dendy, Scott Doebling, and Marianne Francois. And to the administrative staff in those organizations who made it possible for LANL staff to participate remotely, thank you very much, in particular, Rosella Dallman and Pat Rael.

These notes represent about 80% of the course content, the remaining 20% being delivered by in-class demonstrations and a few slideshow presentations. These lectures notes are not designed for dissemination beyond this course.

Please note that references have not yet been added to the lecture notes. In that sense, these notes are not yet complete. Thank you for your understanding in that regard.

Disclaimer

These lecture notes are not suitable for dissemination. They are provided for participants in CU-Boulder CVEN 5838-200 and CVEN 5838-200B, “Computational Multi-Physics Production Software Development.” They have not been put through peer or editorial review.

Contents

Chapter 1

Introduction

The area of computational physics is an emerging discipline that has gradually over the past many decades delivered multiple commercial and industrial grade software products. The educational background of contributors to such products is typically broad, including many of the physical sciences and engineering disciplines, mathematics, and computer science. It is a nontrival endeavor to integrate the knowledge from these different backgrounds, and yet they are often needed in multiple combinations when producing a multi-physics package. Because integrating these disciplines is so demanding, the additional education and training needed to also integrate software engineering processes is typically not considered during the formal education process. As a result, scientists, mathemeticians, and engineers who are involved in production quality scientific software development often must be sent away to additional training courses designed to instill in them the importance of software quality engineering, software project management, verification, and validation. And these training courses must do more than teach about key tools. They are also required to initiate a shift in thinking, a changing of values in the minds of the participants, and a change in culture for the groups they represent. It is a patchwork approach to post-academic training of the individuals involved in engineering these complex products. And yet the scope of that post-academic training is so broad and so important, it deserves the attention of academia and will benefit greatly by being under academic rigor.

This course aims to elevate the discipline of production-quality computational physics software development by doing that. It is the academic setting that has the most influence and authority in providing credentials that employers and auditors acknowledge regarding established skills required to perform tasks they value. By bringing what has been post-academic training into the academic setting and adding to that training the rigors of academia, the subject is elevated and pedigrees of training are established.

Having the performance of students in this course be part of their academic record gives more meaning and depth to the process. But more importantly, it formalizes the language and culture of the engieering discipline. That language and culture has been slow to develop because, for the first several decades after the advent of digital computing, the academic focus of computational physics was on the mathematical, scientific, algorithmic, hardware, and computer science aspects. Advances in those areas are still needed as the breadth of applications continue to grow and the expectations for speed and accuracy grow. But the need for a more firmly established language and culture of product engineering in scientific computing is now evident. Starting about 20 years ago the field began including efforts on on verification, validation, and uncertainty quantification. These aspects, which ultimately judge the efficacy and reliability of the software's output, are helping lead the way to making computational physics codes an intrinsic part of decision making for government agencies and private sector product developers. By adding the academic pursuit of product engineering to those focus areas, the discipline is advanced further.

Design is part of any engineering curriculum. The goal here is to help produce students who are somewhat educated and thereby better qualified to be placed in the production software development environment, having mastered the language of the environment, its culture, and its expectations. In so doing, we advance the rate at which scientific software products bear up under the increasingly heave mantle of reliability, accuracy, and usability.

The harsh reality of the situation, however, is that computational physics is very complex. It is not a subset of an existing discipline. Rather, it brings together multiple disciplines in the ever changing environment of new requirements, new algorithms, new user backgrounds, and new hardware platforms. The discipline requires participation from a wide variety of contributors with rigorous and thorough understanding of their specialties. It is difficult to develop a curriculum that produces an exhaustive knowledge of computational physics, at least one that could be acheivable on a time scale that would make it economically viable for the student. Nevertheless, courses (such as this one) can be added to curricula that provide pedigreed specialization for scientists and others who wish to apply their area of expertise in the production software development environment.

With that in mind, this course has been designed to provide access to and training in the discipline of production quality computational physics software engineering. The technical content of the course is by no means exhaustive. However, it provides some exposure in real and meaningful ways, to some traditionally important concepts in the field. There are many concepts that are not included. But for those coming to the course from other specialties, the technical part of this course is intended to provide some skills that enable them to develop their own software to perform some simulations. For those who already have knowledge in various areas of computational physics, this course hopefully provides exposure to some additional ones and perhaps a reaffirming of the areas they already know.

Chapter 2

Continuum Equations

2.1 Continuum Mechanics

To begin this dicussion, we consider some generic quantity of interest, c, that varies in space and time. It could be a variety of things, including specific internal energy, a chemical species, momentum, or others. There are similarities that drive the movement of c, and so we develop those ideas first without considering exactly what that quantity of interest is. Then we specialize the relationships we obtain for various applications [3].

We are concerned with the distribution of c in space over time, i.e., $c(x_i,t)$, where $i = 1,2,3$ represent the three coordinate directions. The value of $c(x_i,t)$ can change due to multiple factors. Some of them are:

- It can be moved by virtue of the fact that the material in which it resides is moving. Think of c being the concentration of ink in water. If a blob of ink is dropped into a moving stream, c would change due to the fact that the stream is moving.
- It can diffuse.
- In some cases it can be destroyed or created.

We develop the mathematical expressions for these factors and others as well.

2.1.1 The Material Derivative of a Generic Quantity

First, we introduce the concept of a Lagrangian volume V. This volume follows a piece of material as it moves and deforms. No material crosses V's boundary.

We express the rate of change of c inside V using what is sometimes called the "material derivative",

$$\frac{D}{Dt}\int_{V(t)} c dV = \lim_{\Delta t \to 0}\left\{\frac{1}{\Delta t}\left[\int_{V(t+\Delta t)} c(t+\Delta t) dV - \int_{V(t)} c(t) dV\right]\right\}$$

The capital D is used to remind us that the time derivative is for a piece of material that is moving and distorting. While mass does not cross V's boundary, other things can such as magnetic flux lines and heat. We introduce q_i as the flux per unit area per unit time of c through a cross section in space. The increase in c inside V due to it fluxing into it across V's boundaries, which is

$$-\int_V q_i n_i dt \tag{2.1}$$

where n_i is the unit outward normal for each point on V's surface. It is negative because n_i points outward and we are intending here for the flux to result in an increase. Also, here we are using index notation where a repeated subscript, in this case i, implies summation over all coordinate directions. So, to be clear,

$$q_i n_i = \sum_{i=1}^{3} q_i n_i = \mathbf{q} \cdot \mathbf{n} \tag{2.2}$$

In other words, $q_i n_i$ is the dot product between the flux and the surface normal. In addition to q_i flowing across V's sufrace, there could be some type of source term that causes c to be created inside V. We will call that source term S_c, which is on a per-volume basis. Thus, putting the flux and source terms together, we write

$$\frac{D}{Dt}\int_V c dV = \int_S q_i n_i dS + \int_V S_c dV \tag{2.3}$$

From this basic concept, we can derive multiple continuum equations.

2.1.2 Application to the Continuum Mechanics Equations

We consider now the application of the above ideas to the conservation of mass, momentum, and total energy. For the conservation of mass, c in Equation 2.3 is density ρ and also, since mass cannot be created or destroyed, $S_c \equiv 0$. Also, there is by definition no flux of mass crossing the boundary so that $q_i \equiv 0$. Hence Equation 2.3 becomes

$$\frac{D}{Dt}\int_V \rho dV = 0 \tag{2.4}$$

for mass conservation.

For the conservation of momentum, c becomes each of the three momentum components, ρu_j where $j = 1,2,3$. Figuring out what q_i is, in this context, is a bit less intuitive although ultimately there does appear a term that resembles a flux across the surface. Postponing that thought for now, we can instead ask the question, "What surface phenomenon would cause the momentum to change inside V?", or

$$\frac{D}{Dt}\int_V \rho u_j dV = \int_V \; ? \; dS \tag{2.5}$$

This is the continuum form of $F = ma$, and so F, which is the right hand side, must be the integral of stress, which is force per unit area around the surface. In the situation where there are no shear stresses but only a pressure, then we would have simply

$$\frac{D}{Dt}\int_V \rho u_j dV = -\int_V p n_j dS \qquad j = 1,2,3 \tag{2.6}$$

where the negative sign is present because n_j is an outward normal and pressure pushes against the surface. For materials in which there can be shear stresses acting in multiple directions, it is necessary to introduce the more complex stress tensor σ_{ij} that, when dotted with the normal, gives the full stress state at any point on the surface of V. An illustration of the stress tensor is shown in Figure 2.1. Note that it is symmetric. The stress tensor, when dotted with the normal as $\sigma_{ij}n_i$, yields a stress vector in the x_j direction. That stress vector, when integrated around the surface, yields a force. Thus, the more complete desription is

$$\frac{D}{Dt}\int_V \rho u_j dV = \int_V \sigma_{ij} n_i dS \qquad j = 1,2,3 \tag{2.7}$$

Returning now to the idea of a momentum flux, we see that the right-hand side in the above equation somewhat resembles the $q_i n_i$ term in Equation 2.3. The idea of momentum "crossing a boundary" while material does not cross, is a bit counter intuitive, and as done in the sequence above, it is certainly possible to derive the momentum conservation equation without appealing that abstract concept. However, people do from time to time refer to the idea of a "momentum flux," so to help build broader understanding, we make reference to that aspect here.

The concept of energy conservation is useful for evaluating the state of the material, which often depends on its density and specific internal energy (or temperature). One often needs those values as inputs into a constitutive (material response) model that predicts how the material responds to simulus. For example, an equation of state $p = p(\rho, T)$ requires density and temperature to predict the

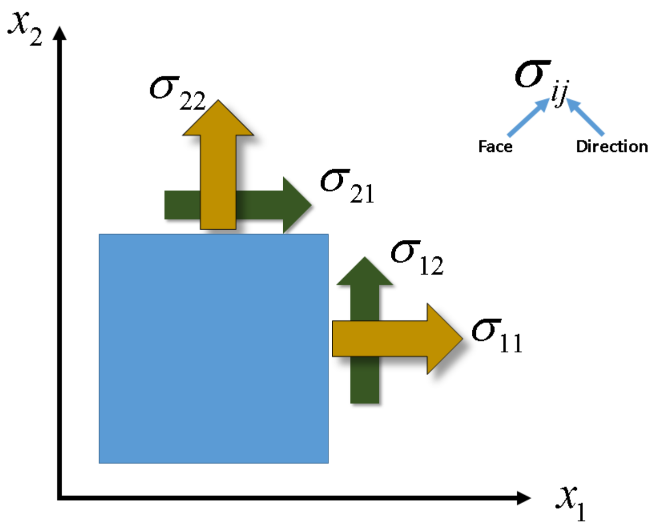


Figure 2.1: The first index of σ_{ij} represents the face number, i.e., which face has a normal in the x_i direction. The second index represents the direction in which the stress acts.

material's pressure, which is used in the momentum equation above either alone or as part of the more complex stress tensor.

To determine specific internal energy (or perhaps temperature), we start with the notion that total mechanical energy E consists of kinetic energy and thermal energy, i.e.,

$$E = \text{I.E.} + \text{K.E.} \tag{2.8}$$

where I.E. on a per-mass basis is the specific internal energy e and K.E. on a per-mass basis is

$$\text{K.E.} = \frac{1}{2}u_i u_i \tag{2.9}$$

So

$$E = \int_V \rho(e + \frac{1}{2}u_i u_i)dV \tag{2.10}$$

For a solid that does not move or deform, only the I.E. term is needed because K.E. is zero. Likewise, for a simulation that does not involve heating, there will be no change in e, so it can be neglected. For a fluid that is moving and heating, both terms are needed.

Both have flux terms as well, one being mechanical work and the other heat other conduction. As with momentum flux, mechanical work can be thought of as a surface flux of energy. And for heat conduction, the heat flux is the flow of specific internal energy per unit time per unit area across the surface. Mechanical work is force times a distance; its rate is force times a velocity, and on a per-area basis it is therefore $(\sigma_{ij}n_i)u_j$.

There are multiple "source" terms that could cause the energy to increase. One could be an external force like gravity, which operates on a per-mass basis. We did not include that term for momentum, so we will not include it here either. Another could be an external heating source, for example if we were to represent heating due to an electromagnetic field. For simplicity, here we will not include that term. But both of those could be added without much difficulty. Proceeding with the remaining terms, we have

$$\frac{D}{Dt}\int_V EdV = \int_S (\sigma_{ij}n_i)u_j dS - \int_S q_i n_i dS \tag{2.11}$$

We note that E is the total mechanical energy, which can be separated into two parts,

$$E = e + \frac{1}{2}u_i u_i \tag{2.12}$$

Here we can harken back to the idea of a flux.

2.1.3 Reynolds Tranport Theorem

The above equations are developed for a volume that moves through space, following a specific mass of material. It can be related to a fixed coordinate system using the Reynold's Tranpsort Theorem, which is derived as follows.

$$\frac{D}{Dt}\int_{V(t)} cdV = \lim_{\Delta t \to 0}\left\{\frac{1}{\Delta t}\left[\int_{V(t+\Delta t)} c(t+\Delta t)dV - \int_{V(t)} c(t)dV\right]\right\}$$

Into this equation, we add and subtract a term that is similar to the ones already there, except unlike those existing terms the times at which the values are evaluated will be mixed. Specifically, we will add and subtract the integral of c over V where V is at time = t, but c is at $t+\Delta t$, i.e., we will add and subtract

$$\int_{V(t)} c(t+\Delta t)dV \tag{2.13}$$

Doing so and gathering terms together produces

$$\frac{D}{Dt}\int_{V(t)} cdV = \lim_{\Delta t\to 0}\left\{\frac{1}{\Delta t}\left[\int_{V(t+\Delta t)} c(t+\Delta t)dV - \int_{V(t)} c(t+\Delta t)dV\right]\right.$$
$$\left. + \frac{1}{\Delta t}\left[\int_{V(t)} c(t+\Delta t)dV - \int_{V(t)} c(t)dV\right]\right\}$$

The second term is the time derviatve for the quantity c inside a *fixed* volume, $V(t)$. So the second term is simply

$$\lim_{\Delta t\to 0}\left\{\frac{1}{\Delta t}\left[\int_{V(t)} c(t+\Delta t)dV - \int_{V(t)} c(t)dV\right]\right\} = \int_{V(t)} \frac{\partial c}{\partial t} \tag{2.14}$$

The first term is more complex to envision. It is the change in the volume integral as the shape of the volume changes, while moving through a steady-state field of c. Imagining a "blob" expanding through a steady state $c(t+\Delta t)$, one can see that the amount the integral would change as it moves from $V(t)$ to $V(t+\Delta t)$ is the amount swept out by the surface at it moves. Considering a small patch of the surface, dS, $c(t+\Delta t)$ is swept out by that face in the amount $\Delta t u_i n_i$, i.e.,the surface must have a component of motion in the normal direction of its surface to sweep out any new volume. Thus, the first term becomes

$$\lim_{\Delta t\to 0}\left\{\frac{1}{\Delta t}\left[\int_{V(t+\Delta t)} c(t+\Delta t)dV - \int_{V(t)} c(t+\Delta t)dV\right]\right\} = \int_{S(t))} c(t)u_i n_i dS \tag{2.15}$$

Putting the above results together gives us a form of the Reynolds Transport Theorem

$$\frac{D}{Dt}\int_{V(t)} cdV = \int_{V(t)} \frac{\partial c}{\partial t}dV + \int_{S(t))} cu_i n_i dS \tag{2.16}$$

One last step is needed to make the relation useful for our applications, application of the Gauss Divergence Theorem to the last term on the right,

$$\frac{D}{Dt}\int_{V(t)} cdV = \int_{V(t)} \frac{\partial c}{\partial t}dV + \int_{V(t)} \frac{\partial cu_i}{\partial x_i}dV \tag{2.17}$$

Bringing all terms on the right into one integral,

$$\frac{D}{Dt}\int_{V(t)} cdV = \int_{V(t)} \left(\frac{\partial c}{\partial t} + \frac{\partial cu_i}{\partial x_i}\right) dV \tag{2.18}$$

This relationship will be applied to the above conservation equations in the next section.

2.1.4 Partial Differential Equation Forms of Mechanical Conservation

Applying the Reynolds Transport Theorem to the conservation of mass equation,

$$\frac{D}{Dt}\int_V \rho dV = 0 \tag{2.19}$$

Here c is identified with ρ, and we get

$$\int_{V(t)} \left(\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} \right) dV = 0 \tag{2.20}$$

Next, we invoke for the first time an argument we will use multiple times later, which is that if the above equation is true for any $V(t)$, it must be true for the integrand on a point-wise basis. Thus, we can write from the above equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0. \tag{2.21}$$

Applying the Reynolds Transport Theorem to the conservation of momentum equation,

$$\frac{D}{Dt}\int_V \rho u_j dV = \int_V \sigma_{ij} n_i dS \tag{2.22}$$

we identify c as ρu_j, for each of the coordinate directions ($j = 1,2,3$), and so using the Reynolds Transport Theorem to convert the left-hand side we write

$$\int_V \left(\frac{\partial \rho u_j}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_i} \right) dV = \int_S \sigma_{ij} n_i dS \tag{2.23}$$

Applying the Gauss Divergence Theorem to the right-hand side,

$$\int_V \left(\frac{\partial \rho u_j}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_i} \right) dV = \int_V \frac{\partial \sigma_{ij}}{\partial x_i} dV \tag{2.24}$$

Again, with the above equation being true for *any* V, the integrands must be equal and so we write

$$\frac{\partial \rho u_j}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_i} = \frac{\partial \sigma_{ij}}{\partial x_i} \tag{2.25}$$

For the total energy equation,

$$\frac{D}{Dt}\int_V E dV = \int_S (\sigma_{ij} n_i) u_j dS - \int_S q_i n_i dS \tag{2.26}$$

we identify c as E to convert the left-hand side, writing

$$\int_V \left(\frac{\partial E}{\partial t} + \frac{\partial E u_i}{\partial x_i} \right) dV = \int_S (\sigma_{ij} n_i) u_j dS - \int_S q_i n_i dS \tag{2.27}$$

It is necessary to apply the Gauss Divergence Theorem to the two surface integrals on the right hand side in order to have every term expressed as volume integrals. Doing that, we write

$$\int_V \left(\frac{\partial E}{\partial t} + \frac{\partial E u_i}{\partial x_i} \right) dV = \int_V \frac{\partial \sigma_{ij} u_j}{\partial x_i} dV - \int_V \frac{\partial q_i}{\partial x_i} dV \tag{2.28}$$

As with the other equations, the above equation is true for any V, meaning the integrands must be equal so that

$$\frac{\partial E}{\partial t} + \frac{\partial E u_i}{\partial x_i} = \frac{\partial \sigma_{ij} u_j}{\partial x_i} - \frac{\partial q_i}{\partial x_i} \tag{2.29}$$

The above equation is of limited utility. We have no need for knowing how the total energy of the system evolves, except perhaps for diagnostic purposes. However, we do need to know how the specific internal energy evolves so that we can model how the material is responding to the forces being applied to it. If we dot the momentum equation with $(1/2)u_i$, a scalar equation is produced that has the kinetic energy part of E in it. Subtracting that scalar equation from the total energy equation above produces the evolution equation for specific internal energy,

$$\frac{\partial \rho e}{\partial t} + \frac{\partial \rho e u_i}{\partial x_i} = \frac{\partial \sigma_{ij} u_j}{\partial x_i} - \frac{\partial q_i}{\partial x_i} \tag{2.30}$$

To recap, the three equations for continuum mechanics, conservation of mass and momentum, and the specific internal energy equation, are

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0 \tag{2.31}$$

$$\frac{\partial \rho u_j}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_i} = \frac{\partial \sigma_{ij}}{\partial x_i} \qquad i = 1,2,3 \tag{2.32}$$

$$\frac{\partial \rho e}{\partial t} + \frac{\partial \rho e u_i}{\partial x_i} = \frac{\partial \sigma_{ij} u_j}{\partial x_i} - \frac{\partial q_i}{\partial x_1} \tag{2.33}$$

2.1.5 The Eulerian Viewpoint Derivation

In this section, we give a partial derivation of the mechanics transport equations using an Eulerian point of view. The results are similar to those above, but we make a few simplifying assumptions along the way in this section.

Consider a small rectangle in the $x-y$ plane that is Δx by Δy in dimension and is aligned with the x and y axes. We seek to quantify the rate of growth of c in that cell, i.e., we seek to develop expressions for

$$\frac{\partial c}{\partial t} \tag{2.34}$$

One reason why the rate of growth of c can be non-zero is because there is an imbalance in how much of c is entering or leaving the cell. We express the movement of c through space as the "flux" of c, and it is expressed as a per-time, per-area quantity. Movement implies direction, and so flux is a vector quantity.

Let the flux of c be q, where

$$\mathbf{q} = q_x\mathbf{i} + q_y\mathbf{j} \tag{2.35}$$

The flux entering the left-facing part of our cell is

$$q_{x-l} \tag{2.36}$$

while the flux leaving the right-facing part is

$$q_{x-r} \tag{2.37}$$

Similarly, the flux flowing into the bottom face and leaving the upper face are

$$q_{y-b} \text{ and } q_{y-t} \tag{2.38}$$

Thus, we can write that

$$\frac{\partial c \Delta x \Delta y}{\partial t} = q_{x-r}\Delta y - q_{x-l}\Delta y + q_{y-b}\Delta x - q_{y-t}\Delta y \tag{2.39}$$

The Δx and Δy terms are needed on the left-hand side because c is volume based and they are needed on the right-hand side because $\mathbf{q}$ is area based. We express the flux leaving the right face using a Taylor series expansion about the left face, i.e, we say

$$q_{x-r} = q_{x-l} + \frac{\partial q_x}{\partial x}\Delta x \tag{2.40}$$

Likewise, in the $y-$ direction, we write

$$q_{y-t} = q_{y-b} + \frac{\partial q_y}{\partial y}\Delta y \tag{2.41}$$

Upon substitution into Equation 2.39 we obtain

$$\frac{\partial c\Delta x\Delta y}{\partial t} = -\frac{\partial q_x}{\partial x}\Delta y\Delta x - \frac{\partial q_y}{\partial y}\Delta x\Delta y \tag{2.42}$$

Another way c can increase is through some type of source term that causes c to be injected into the situation. We represent this source term as Q, on a per-volume basis. So, including the source term, we have

$$\frac{\partial c\Delta x\Delta y}{\partial t} = -\frac{\partial q_x}{\partial x}\Delta y\Delta x - \frac{\partial q_y}{\partial y}\Delta x\Delta y + Q\Delta x\Delta y \tag{2.43}$$

The Δ terms cancel leaving

$$\frac{\partial c}{\partial t} = -\left(\frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y}\right) + Q \tag{2.44}$$

or

$$\frac{\partial c}{\partial t} = -\nabla \cdot \mathbf{q} + Q \tag{2.45}$$

Causes for flux: Diffusion

Diffusion is a phenomenon in nature that causes our quantity of interest, c, to flux through our continuum. In diffusion, the flux of c is due to a gradient in the value of c. Specifically, if c is higher in one location than in a neighboring location, some of c will move from the higher concentration location into the lower concentration location. In the $x-$ and $y-$ directions, diffusive flux is proportional to the spatial slope of c in those two directions, specifically,

$$q_x = -k\frac{\partial c}{\partial x}$$
$$q_y = -k\frac{\partial c}{\partial y}$$

where k is the proportionality factor. The k value need not be a constant, nor does it have to be the same for the $x-$ and $y-$ directions. Here, just for simplicity, we consider an isotropic material, i.e., one k for both directions.

In vector form, then,

$$\mathbf{q} = -k\nabla c \tag{2.46}$$

Substituting Equation 2.46 into Equation 2.45, we get

$$\frac{\partial c}{\partial t} = \nabla \cdot (k\nabla c) + Q \tag{2.47}$$

This is a partial differential equation (PDE) in c known as the diffusion equation.

Causes for flux: Convection

Another way in which c can move through space is if the material in which it resides moves. For instance, if there is no diffusion but the material is moving, c is transported according to the velocity field. In this case, we write

$$q_x = u_x c$$
$$q_y = u_y c$$

where u_x and u_y are the $x-$ and $y-$ components of velocity. Or, in vector notation,

$$\mathbf{q} = \mathbf{u}c \tag{2.48}$$

Substituting Equation 2.48 into Equation 2.45, we get

$$\frac{\partial c}{\partial t} = -\nabla \cdot (\mathbf{u}c) + Q \tag{2.49}$$

Convection-Diffusion Equation

It is possible for c to be transported by convection and diffusion at the same time. Consider, for example, using a medicine dropper to drop a blob of ink into a flowing stream. The ink is transported by the stream's velocity. But if the water is sufficiently warm, the ink also diffuses. In this case, the fluxes from each are additive, and in to accommodate that we have the convection-diffusion equation

$$\frac{\partial c}{\partial t} = \nabla \cdot (k\nabla c) - \nabla \cdot (\mathbf{u}c) + Q \tag{2.50}$$

Applications of the Convection-Diffusion Equation

If in Equation 2.50 we let c be mass per unit volume, i.e., density, we obtain the equation for mass conservation. Bulk mass is not transported by diffusion and there is no source term for mass, so only the convective term remains, and we write

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\mathbf{u}\rho) \tag{2.51}$$

Compare this to the same equation derived using the Lagrangian viewpoint and Reynold's Transport Theorem, Equation 2.31.

In the case of momentum, c is momentum per unit mass, or $\rho\mathbf{u}$. While this is a vector quantity, we may consider each direction separately. And in this case, we must consider a source term. While there is no source term for bulk mass, there can be a source term for momentum, i.e., something acting on our Δx by Δy cell that causes momentum to increase. In particular, force can act on our cell.

Consider the $x-$ component of the momentum equation, i.e., where c is ρu_x and there are no viscous effects, which cause momentum to diffuse. Without viscosity, we only have the convection term and a momentum source term,

$$\frac{\partial \rho u_x}{\partial t} = -\nabla \cdot (\mathbf{u}u_x\rho) + Q_x \tag{2.52}$$

In the simplest of situations, the source for momentum is pressure acting on each face of our cell. The force acting on the left face is $p_l\Delta y$ and the pressure acting on the right is pushing in the negative direction and is $-p_r\Delta y$. Like with flux, we extrapolate for p_r,

$$p_r = p_l + \frac{\partial p}{\partial x}\Delta x \tag{2.53}$$

Thus the net force in the $x-$ direction is

$$(p_r - p_l)\Delta y = -\frac{\partial p}{\partial x}\Delta x\Delta y \tag{2.54}$$

and so the per-volume source term here is is $-\frac{\partial p}{\partial x}$. Thus, we write

$$\frac{\partial \rho u_x}{\partial t} = -\nabla \cdot (\mathbf{u}u_x\rho) - \frac{\partial p}{\partial x} \tag{2.55}$$

and similarly for the $y-$ component of velocity,

$$\frac{\partial \rho u_y}{\partial t} = -\nabla \cdot (\mathbf{u}u_y\rho) + \frac{\partial p}{\partial y} \tag{2.56}$$

Or in vector form,

$$\frac{\partial \rho \mathbf{u}}{\partial t} = -\nabla \cdot (\mathbf{u}\mathbf{u}\rho) - \nabla p \tag{2.57}$$

Compare this to the momentum equation derived using the Lagrangian viewpoint and Reynold's Transport Theorem, Equation 2.32. Although here, the stress tensor is simply the pressure term.

2.1.6 Mechanics Constitutive Models

The above equations have the following unknowns:

$$\text{3 velocity components: } u_i \qquad i = 1,2,3 \tag{2.58}$$

$$\text{density} \rho \tag{2.59}$$

$$\text{6 components of the stress tensor:} \sigma_{ij} \tag{2.60}$$

where the 6 in the last statement is due to symmetry, i.e., $\sigma_{ij} = \sigma_{ji}$. So in total, there are 10 unknowns. And we only have 5 equations (recall the momentum equation is for $i = 1,2,3$.

There are no additional physical laws to employ to further constrain the system of equations. Rather, the constraints come from models for the particular material to which the equations are applied. These models are called "constitutive models", because they express what constitutes the materials being simulated.

Constitutive models connect how materals react to stimulus or disturbances. In this particular example of continuum mechanics, the material reacts by altering its stress tensor, and it alters that tensor based on how it is being deformed.

Fluids

In the above discussion, we have assumed velocity plays a central role in the mechanics of the situation. We shall shortly see that the above equations apply to solids as well. But for now, we focus on the idea of constitutive models for fluids.

Fluids can be divided into two categories: Gases and liquids. Gases are defined as those fluids that cannot produce any type of stress other than simple pressure. Pressure is a stress that acts uniformly in all directions. A gas' pressure increases with its density and its specific internal energy. An equation of state ("EOS") is used to provide pressure as a function of those two variables, i.e., $p = p(\rho, e)$. Even though that relationship is often called an EOS, oftentimes it is actually a tabular lookup. In any case, for gases

$$\sigma_{ij} = -\begin{bmatrix} p & 0 & 0 \\ 0 & p & 0 \\ 0 & 0 & p \end{bmatrix} \tag{2.61}$$

Here, the stress tensor reduces to pressure, bringing the total number of unknowns down to 6. With the addition of the EOS for pressure, there are six equations and the system is closed.

For liquids, the situation is different, and in some ways, more complicated. Firstly, for liquids it is typically assumed that their density is relatively constant. This assumption is referred to as "incompressible flow". In that situation, density is now a known value, an initial condition. However, we no longer have an EOS that provides pressure. Rather, pressure is a unknown governed by the conservation equations, like velocity. Also, liquids, by definition, can produce more complex stresses. In particular, they can produce a shear stress. In general, that type of shear stress can exist in all three coordinate directions, each time being caused by a gradient in the velocity fields. The resulting constitutive model for what is called a "Newtonian" liquid, the simplist kind, is

$$\sigma_{ij} = -p\delta_{ij} + \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.62}$$

where δ_{ij} is the Kronecker delta, which is equal to unity if $i = j$ and zero otherwise. So this situation is very different than for gases. The stress tensor depends heavily on the "kinematic" unknowns (velocity) instead of just the thermodynamic state, and pressure is an unknown value for which we have no EOS.

But if we re-count the number of equations and unknowns, we see that the system is closed. We have the mass conservation equation and three momentum equations, resulting in four equations; note that we do not need the specific internal energy equation, since we do not have an EOS that needs e. And we only have four unknowns: The three velocity components and pressure. So, again, the system is closed even though its form is very different than the resulting equations for gases.

2.1.7 Elastic Deformation

If we are performing a static analysis of solids that haved deformed very small amounts, we can set velocity to zero in the governing equations. Conservation of mass is automatically satisfied under that scenario and the momentum equations reduce to

$$\frac{\partial \sigma_{ij}}{\partial x_i} = 0 \qquad i = 1,2,3 \tag{2.63}$$

In 2-D Cartesian coordinates, using $x - y$ notation, they are

$$\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} = 0 \tag{2.64}$$

$$\frac{\partial \sigma_{yx}}{\partial y} + \frac{\partial \sigma_{yy}}{\partial y} = 0 \tag{2.65}$$

Remember that $\sigma_{xy} = \sigma_{yx}$. In this case, we have two governing equations and three unknowns, again, a constitutive model is needed. For small-scale elastic deformation, the multi-dimensional form of what is called "Hook's Law" is used, even though it is only a model not actually a law.

Velocities are basically zero, but deformation is not. Here, we let physical displacement of any point be denoted by the vector $v_i, i = 1,2,3$. Strain is a non-dimensionalized version of displacement. Think of it as a percentage based quantity, for example, the percent a rod is stretched or compressed. We express the strain as derivatives of the displacement field as follows.

Consider two adjacent points, A and B, separated by dx_1 along the x_1- axis as shown in Figure 2.2. Both have a displacement when the material in which they reside is stressed. Let v_1 represent the displacement of any point in the x_1 direction. The growth in the dstance between A and B after distortion is

$$v(B)_1 - v(A)_1 \tag{2.66}$$

Using a Taylor series expansion to represent B's displacement in terms of A's,

$$v(B)_1 = v(A)_1 + \frac{\partial v_1}{\partial x_1}\Delta x_1 \tag{2.67}$$

where Δx_1 was the original distance between A and B. Using that Taylor series expansion, the growth in the distance (2.66) is

$$\frac{\partial v_1}{\partial x_1}\Delta x_1 \tag{2.68}$$

In keeping with the idea that we want a percentage-based measure of deformation, we divide the amount of stretching by the original distance dx_1. In so doing, we

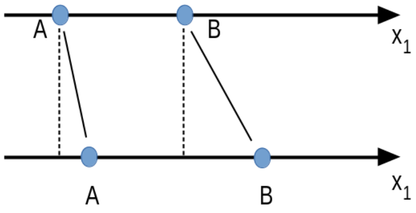


Figure 2.2: Two points before (top) and after (bottom) deformation.

define normal strain in the x_1 direction as

$$\varepsilon_{11} = \frac{\partial v_1}{\partial x_1} \tag{2.69}$$

and similarly for the other normal directions.

We also track a non-dimensional form of angular deformation, how much a right-angle is distorted. We define a metric that rates the relative motion of two opposing corners of a square in the $x_1 - x_2$ plane as the shear strain,

$$\varepsilon_{12} = \frac{\partial v_1}{\partial x_2} + \frac{\partial v_2}{\partial x_1} \tag{2.70}$$

and similarly for the other directions.

Hook's Law relates the strain tensor ε_{ij} to the stress tensor σ_{ij}, and it comes in many forms. One way is to first define the bulk volumetric strain and then deviations from that by separating the strain tensor as follows:

$$\varepsilon_{ij} = \begin{bmatrix} V & & \\ & V & \\ & & V \end{bmatrix} + \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \tag{2.71}$$

where

$$V = \frac{1}{3}\varepsilon_{kk} \tag{2.72}$$

then Hook's Law may be written as

$$\sigma_{ij} = 3KV\delta_{ij} + 2Ge_{ij} \tag{2.73}$$

where K is the bulk modulus and G is the shear modulus.

2.1.8 Solids under Large Deformation

When solids are subjected to large deformation, the constitutive laws must accomodate the notion that the material yields, i.e., it deforms so much that even if the forces on it are removed, it will not recover its original shape. To deform in such a way that the material does not fully recover is called "plastic deformation", where the term "plastic" refers to the permanence of the deformation. The field of study for developing constitutive models for such situations is called "plasticity", and is a topic that cannot be fully covered in this course. However, we can still lay out the most basic ideas to provide a foothold in discussions. It is important that the reader understand the topic is broader and more complex than is covered here.

In studying plasticity, we separate the stress tensor into two parts:

$$\sigma_{ij} = -\begin{bmatrix} p & & \\ & p & \\ & & p \end{bmatrix} + \begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{bmatrix} \tag{2.74}$$

The first part is the bulk pressure, and just as in the case for a gas, we expect to have an EOS for it. The second part is called the "deviatoric" stress tensor; it represents deviations from the bulk pressure. It turns out that bulk pressure does not typically result in plastic deformation; plastic deformation is often only attributable to the deviatoric stresess, a fact that is also borne out mathematically. One notable exception is the plastic deformation of pores, which can occur due to bulk pressure.

Setting that exception aside for now, we seek to know how the deviatoric stresses respond in a material subjected to large-scale deformation. In the 1930s, the Prandtl-Reuss model was developed, which has proven very effective. One very important aspect of the model is that it is not a plug-and-play equation with simple input and output. Rather, it is a time-varying ordinary differential equation that must be evolved with transient governing equations. The form of the model is

$$\dot{s}_{ij} = 2G\dot{e}_{ij} - \frac{G}{Y^2}\left(\dot{s}_{mn}e_{mn}\right)s_{ij} \tag{2.75}$$

where Y is the yield stress measured in the laboratory. It was also discovered that the invariant

$$J_2 = \frac{1}{6}\left[(\sigma_{11}^2 - \sigma_{22}^2) + (\sigma_{22}^2 - \sigma_{33}^2) + (\sigma_{33}^2 - \sigma_{11}^2)\right] + \sigma_{12}^2 + \sigma_{13}^2 + \sigma_{23}^2 \tag{2.76}$$

provides an excellent metric to determine when yielding occurs. The value of J_2 does not change if the coordinate system is rotated; it is one of the scalar "invariants" that represent the magnitude of the stress tensor. When J_2 is sufficiently small compared to Y, the material behaves elstically, according to Hooke's Law. Otherwise, the stress evolves according to the Prandtl-Reuss equation.

Note that Y can change with strain history and also with temperature.

2.2 Electromagnetics

The equations governing electromagnetic fields in space-time are Maxwell's equations. We develop those equations, informally, here.

We start by defining a field **D**, which we will call the "electric field density" that is due to charge Q centered at $(x,y,z) = (0,0,0)$ as follows:

$$\mathbf{D} \equiv \frac{Q}{4\pi r^2}\hat{\mathbf{r}} \tag{2.77}$$

where $\hat{\mathbf{r}} = \hat{\mathbf{r}}(x,y,z)$ is a unit vector pointing from Q to the point (x,y,z) under consideration, and r is distance between that point and Q. Notice that the field density decreases with distance and that its direction is always pointing away from the source, Q, as long as Q is positive. We have not yet shown why **D** is a definition that has physical relevance, but we will shortly. It can be shown mathematically that for any simply connected, unbroken surface S fully enclosing Q, that

$$\int_S \mathbf{D}\cdot\mathbf{n}dS = Q \tag{2.78}$$

This equation is the first of Maxwell's equations, and it is referred to as Gauss' Law. Its relevance to electromagnetism will be shown shortly.

Next, we consider magnetic fields, specifically, the analogy to **D**, which we will call **B**, the magnetic flux density. It has been observed that, unlike electric fields, magnetic fields do not have single sources that cause the fields that emenate from them. In other words, they have no Q analogy. Whether magnetic fields are due to magnets or due to the flow of electric current, the magentic field lines are closed. They do not "eminate outward' as do electric fields. Thus for magentic fields, **B** is *solenoidal* and since it is solenoidal we can write

$$\int_S \mathbf{B}\cdot\mathbf{n}dS = 0 \tag{2.79}$$

which is Maxwell's second equation. It is an observation about magnetic flux lines and the observation that a magnetic monopole has never been observed.

These first two equations are conservation equations in the sense that the relate properites of these two fields **D** and **B** as we have defined them to their sources in a way that is invariant. Before moving to the next two Maxwell equations, let us now discuss why **D** has relevance. It has been shown experimentally, that an electric charge, Q, placed at the origin, will impose a force **F** on another electric charge q in an amount that is inversely proportional to the distance between them squared, and also proportional to the inverse of the permittivity or dielectric constant, ε,

$$\mathbf{F} = q\frac{Q}{4\pi r^2 \varepsilon}\hat{\mathbf{r}} \tag{2.80}$$

The medium matters in the physics here, a fact that is represented by the presence of ε. The material matters because materials vary in how well they accomodate an electric field. For example, the existence of an electric field in a material may cause the molecules of the material to polarize, setting up a competing electric field in the opposite direction.

The electric field is defined as the force **F** due to **E** per unit charge, i.e.,

$$\mathbf{E} \equiv \mathbf{F}/q \tag{2.81}$$

To be clear, then

$$\mathbf{E} = \frac{Q}{4\pi r^2 \varepsilon}\hat{\mathbf{r}} \tag{2.82}$$

It is helpful to define a quantity that is independent of the particular material, which allows us to write the governing equations in a material-independent mannar, and that is why we defined **D** above. Note that

$$\mathbf{D} = \varepsilon \mathbf{E} \tag{2.83}$$

Now we consider a third Maxwell equation. It has been found experimentally that a wire carrying a current I, when placed in a magnetic field has a force exerted on it. That force is of the form

$$F = L \cdot I \cdot B \tag{2.84}$$

where L is the length of wire in the field and B is related to the magnetic field. So B is in Newtwons per Ampere per meter. The force is a result of the fact that when the current flows through the wire it creates a magnetic field that interacts with that of the magnet, together resulting in the force.

Removing, now, the wire from the field of the magnet, we can measure that field by measuring the torque on a small magnetic needle placed at different radii from the wire. It is found that

$$B = \frac{\mu I}{2\pi r} \tag{2.85}$$

where μ is the magnetic permittivity of the medium and r is the radius outward from the wire. The current I has a direction and the direction of the field B is tangent to the circular path around I, where that tangential direction $\mathbf{t}$. So, in vector form

$$\mathbf{B} = \left(\frac{\mu I}{2\pi r}\right)\hat{\mathbf{t}} \tag{2.86}$$

Now, if we integrate the above equation along a circular line integral, we get

$$\begin{aligned}
\int_c \mathbf{B}\cdot d\mathbf{l} &= \int_c \left(\frac{\mu I}{2\pi r}\right)\mathbf{t}\cdot d\mathbf{l} \\
&= \left(\frac{\mu I}{2\pi r}\right)\int_c \mathbf{t}\cdot d\mathbf{l} \\
&= \left(\frac{\mu I}{2\pi r}\right)2\pi r \\
&= \mu I
\end{aligned}$$

Moreover, the above relationship is linear and so we can use superposition. There can be several wires with different currents and we can add their effects linearly. Even more generally, we can replace the idea of **I** (in Amperes) flowing through a wire with the idea of a current flux density **J** (in Amperes per square meter) flowing through an area, A, and write the above result as

$$\int_c \mathbf{B}\cdot d\mathbf{l} = \mu \int_A \mathbf{J}\cdot \mathbf{n} dS \tag{2.87}$$

One of Maxwell's important contributions is that he recognized the above equation is incomplete. One way to reveal the problem with the above equation is by allowing the surface A to be very distorted such that it is no longer simply an circle orthogonal to a wire, but rather, a bulb shaped surface that never crosses the wire and, instead, passes through the space between two surfaces of a capacitor to which the wire is connected. In the situation where the capacitor is being charged, current is flowing in the wire, inducing a **B** field as usual, but because of the way we chose A, no current is flowing through A. However, what *is* occuring is an electric field is being set up in the capacitor as it charges. Thus Maxwell solved what was known at that time as“the capacitor problem” by adding an additional term, which he called the “displacement current”. The final form of Ampere’s law, one of the four Maxwell equations, is

$$\int_c \mathbf{B}\cdot d\mathbf{l} = \mu\left(\int_A \mathbf{J}\cdot\mathbf{n}dS + \varepsilon\int_A \frac{\partial\mathbf{E}}{\partial t}\cdot\mathbf{n}dS\right) \tag{2.88}$$

Note that there are plenty of situations wherein we have no current, i.e., $\mathbf{J}=\mathbf{0}$ and all we have is fluctuations in the electric field, e.g., electromagnetic waves.

The final Maxwell equation is the analogy of the one described above, which stated that a flowing current or changing **E** field causes a magnetic field. And now, this last equation known as Faraday’s Law, quantifies the observation that a moving magnetic field can induce a current. In other words, there is a reciprocal relationship, of sort with the previous equation.. The observation was made independently by Michael Faraday in London and Joseph Henry in Albany in about 1831 that if a magnet was moved in and out of the center of a coil of wire a current was induced. That current must be due to an electric field around that coil. In a mannar similar to the simple version of Ampere’s Law, generalized to surface areas an line integrals of any shape, the final Maxwell equation is

$$\int_c \mathbf{E}\cdot d\mathbf{l} = -\frac{d}{dt}\int_A \mathbf{B}\cdot\mathbf{n}dS \tag{2.89}$$

In summary, the governing equations for electromagnetic fields, Maxwell’s equations using the constitutive laws for simple materials (isotropic, homogeneous, non-frequency dependent) are stated in the table. At the same time, by the application of Stoke’s theorem to the line integrals, it is possible to express all terms in the last two equations as area integrals. Using the same type of argument used for continuum mechanics, where if the integrals must be equal for any volume the integrands must be equal, here the argument is made for arbitrary *areas*. The result is PDE forms, which are expressed alongside the integral forms derived above; note that in the process we introduce charge density ρ, where

$$Q = \int_V \rho dV \tag{2.90}$$

The electric field can be expressed as the gradient of a potential; here that potential is voltage, V. Hence

$$\mathbf{E} = -\nabla V \tag{2.91}$$

Law	Integral Form	PDE Form
Gauss	$\int_S \mathbf{E}\cdot\mathbf{n}dS = \frac{1}{\varepsilon}Q$	$\nabla\cdot\mathbf{E} = \rho/\varepsilon$
Gauss Magnetic	$\int_S \mathbf{B}\cdot\mathbf{n}dS = 0$	$\nabla\cdot\mathbf{B} = 0$
Ampere	$\int_c \mathbf{B}\cdot d\mathbf{l} = \mu\left(\int_A \mathbf{J}\cdot\mathbf{n}dS + \varepsilon\int_A \frac{\partial\mathbf{E}}{\partial t}\cdot\mathbf{n}dS\right)$	$\nabla\times\mathbf{B} = \mu\left(\mathbf{J} + \varepsilon\frac{\partial\mathbf{E}}{\partial t}\right)$
Faraday	$\int_c \mathbf{E}\cdot d\mathbf{l} = -\frac{d}{dt}\int_A \mathbf{B}\cdot\mathbf{n}dS$	$\nabla\times\mathbf{E} = -\frac{d}{dt}\mathbf{B}$

so that Gauss' Law may be expressed as

$$\nabla^2 V = \frac{\rho}{\varepsilon} \tag{2.92}$$

Chapter 3

Finite Difference Method

3.1 Diffusion Equation Application

We start with the transient diffusion equation

$$C\frac{\partial \phi}{\partial t} - \frac{\partial}{\partial x_i}\left(D\frac{\partial \phi}{\partial x_i}\right) = q \tag{3.1}$$

For constant D in 2-D,

$$C\frac{\partial \phi}{\partial t} - D\left(\frac{\partial^2 \phi}{\partial x_1^2} + \frac{\partial^2 \phi}{\partial x_2^2}\right) = q \tag{3.2}$$

We will consider orthogonal grids, aligned with the $x_1 - x_2$ axes. And since we are only dealing with scalars here, it is easier to adopt the use of $x - y$ in place of $x_1 - x_2$. Thus, we write

$$C\frac{\partial \phi}{\partial t} - D\left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2}\right) = q \tag{3.3}$$

Consider the grid shown in Figure 3.1. On that grid we have defined discrete values of ϕ using logical indexing for the $x-$ and $y-$ directions. We can estimate the partial derivative of ϕ with respect to x on a vertical face using adjacent values, i.e.,

$$\left.\frac{\partial \phi}{\partial x}\right|_{i+1/2,j} \approx \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x} \tag{3.4}$$

Similarly, for the vertical face on the other side of $\phi_{i,j}$

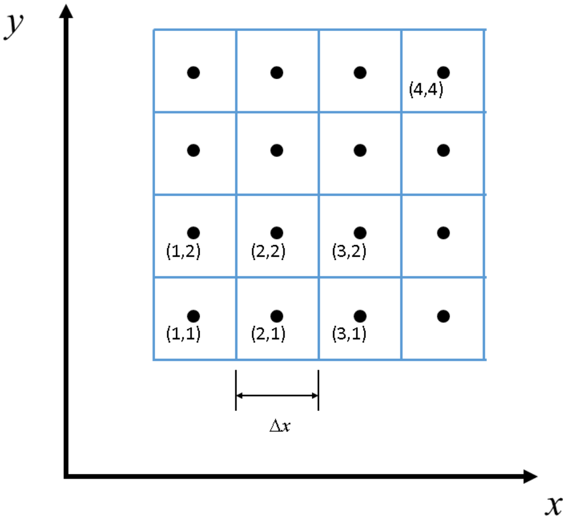


Figure 3.1: Two-dimensional finite difference grid with logical i-j numbering.

$$\left.\frac{\partial \phi}{\partial x}\right|_{i-1/2,j} \approx \frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta x} \tag{3.5}$$

The second derivative can be approximated by taking the derivative of the first derivative, so we write

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\left.\frac{\partial \phi}{\partial x}\right|_{i+1/2,j} - \left.\frac{\partial \phi}{\partial x}\right|_{i-1/2,j}}{\Delta x} \tag{3.6}$$

Substituting the above approximations, we write

$$\begin{aligned}\frac{\partial^2 \phi}{\partial x^2} &\approx \frac{\frac{\phi_{i+1,j}-\phi_{i,j}}{\Delta x} - \frac{\phi_{i,j}-\phi_{i-1,j}}{\Delta x}}{\Delta x} \\ &= \frac{\phi_{i+1,j} - 2\phi_{i,j} - \phi_{i-1,j}}{\Delta x^2}\end{aligned}$$

We can perform an analogous operation in the $y-$ direction, and get

$$\frac{\partial^2 \phi}{\partial y^2} \approx \frac{\phi_{i,j+1} - 2\phi_{i,j} - \phi_{i,j+1}}{\Delta y^2}$$

For simplicity, we will let $\Delta x = \Delta y \equiv h$, so that the above second-derivative approximations can be combined:

$$\begin{aligned}\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} &\approx \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{h^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j+1}}{h^2} \\ &= \frac{\phi_{i+1,j} + \phi_{i,j+1} - 4\phi_{i,j} + \phi_{i+1,j} + \phi_{i,j+1}}{h^2} \end{aligned} \tag{3.7}$$

Returning now to the transient diffusion, we are almost ready to substitute Equation 3.7 into the equation we are trying to solve, Equation 3.2. However, we first must determine how to deal with the transient term.

3.1.1 Forward Euler Temporal Discretization

We introduce a finite difference approximation for the transient term,

$$\frac{\partial\phi}{\partial t} \approx \frac{\phi^{n+1} - \phi^n}{\Delta t} \tag{3.8}$$

where time, $t^{n+1} = t^n + \Delta t$ and we have adopted the use of superscripts to denote the time at which the variable is evaluated. We need to tie the above idea to our spatial discretization as well, though, and so we combine the n superscript with the i,j subscript to write, for ϕ at grid point $i-j$,

$$\frac{\partial\phi_{i,j}}{\partial t} \approx \frac{\phi_{i,j}^{n+1} - \phi_{i,j}^n}{\Delta t} \tag{3.9}$$

So, now we have discrete solution values at each grid point i,j, through time, $t^n, n = 1,2,3,....$

A decision must be made now regarding how to use Equations 3.9 and 3.7 in Equation 3.2. Specifically, when we developed Equation 3.7 we temporarily overlooked the fact that the solution must be transient. We had not introduced the concept of the n superscript. However, in combining these terms together, it is necessary to choose where, in time, the terms in Equation 3.7 are evaluted.

That choice has important implications. If we choose to evaluate the terms in Equation 3.7 at timestep n, i.e., t^n, we obtain the following:

$$\frac{\phi_{i,j}^{n+1} - \phi_{i,j}^n}{\Delta t} - D\frac{\phi_{i+1,j}^n + \phi_{i,j+1}^n - 4\phi_{i,j} + \phi_{i+1,j}^n + \phi_{i,j+1}^n}{h^2} = q(t^n) \tag{3.10}$$

We use the above equation to move the solution values from t^n to t^{n+1}. In other words, we assume we know $\phi^n_{i,j}$ for all i, j. In the beginning of the problem, at $n = 0$, this knowledge is called the “initial condition”. The problem must come with an initial condition or we would not know how to solve it.

Returning to Equation 3.10, we can solve for $\phi^{n+1}_{i,j}$ to get

$$\phi^{n+1}_{i,j} = \phi^n_{i,j} + \Delta t D \frac{\phi^n_{i+1,j} + \phi^n_{i,j+1} - 4\phi_{i,j} + \phi^n_{i+1,j} + \phi^n_{i,j+1}}{h^2} + \Delta t q(t^n) \tag{3.11}$$

or

$$\phi^{n+1}_{i,j} = \phi^n_{i,j} + \frac{\Delta t D}{h^2}\left(\phi^n_{i+1,j} + \phi^n_{i,j+1} - 4\phi_{i,j} + \phi^n_{i+1,j} + \phi^n_{i,j+1}\right) + \Delta t q(t^n) \tag{3.12}$$

When referring to the temporal discretization aspects of the above formulation, it is called a “Forward Euler” scheme. The idea of Forward Euler time discretization can be applied to other spatial discretization methods, e.g., finite elements. The important idea is that the spatial discretization is evaluated at the “old” time step, so it is known.

While the Forward Euler method seems very straightforward (and it is), it does have drawbacks. Specifically, because we are only using information from the previous time step to propel the solution forward in time, there are limitations on how big of a timestep we can take. In reality, the spatial derivatives will change based on how the solution evolves. If we use a timestep that is too large, we are in effect holding those spatial derivatives fixed from the previous timestep, not allowing them to change with the situation. The result is worse than a loss of accuracy; there can be a loss of stability. This aspect will be discussed further, later. Still, however, despite its timestep limitatioins, the Forward Euler method remains very popular in computational mechanics codes.

3.1.2 Backward Euler Temporal Discretization

As an alternative to the Forward Euler method, we could evaluate the spatial derivative terms at the *new* timestep, i.e., we could write

$$C\frac{\phi^{n+1}_{i,j} - \phi^n_{i,j}}{\Delta t} - D\frac{\phi^{n+1}_{i+1,j} + \phi^{n+1}_{i,j+1} - 4\phi^{n+1}_{i,j} + \phi^{n+1}_{i+1,j} + \phi^{n+1}_{i,j+1}}{h^2} = q(t^n) \tag{3.13}$$

In this formulation, almost none of the terms are known. We have $\phi^n_{i,j}$ on the left-hand side that is known from the previous timestep. Other than that, all the remaiining terms are unknown.

However, the above equation must hold true for all points *simultaneously*. We rearrange Equation 3.13 so that all of the known terms are on the right-hand side and the unknown terms are kept on the left-hand side.

$$\left(C+4\Delta tD/h^2\right)\phi_{i,j}^{n+1}-\left(\Delta tD/h^2\right)\left(\phi_{i+1,j}^{n+1}+\phi_{i,j+1}^{n+1}+\phi_{i+1,j}^{n+1}+\phi_{i,j+1}^{n+1}\right)=C\phi_{i,j}^{n}+q\Delta t \tag{3.14}$$

Defining

$$\bar{C}\equiv\left(C+4\Delta tD/h^2\right)$$
$$\bar{D}\equiv\left(\Delta tD/h^2\right)$$

Equation 3.14 looks simpler. We will write it using $\bar{C}$ and $\bar{D}$ and, also, show more explicitly that it must be true for all the points:

$$\bar{C}\phi_{i,j}^{n+1}-\bar{D}\left(\phi_{i+1,j}^{n+1}+\phi_{i,j+1}^{n+1}+\phi_{i+1,j}^{n+1}+\phi_{i,j+1}^{n+1}\right)=\bar{C}\phi_{i,j}^{n}+q\Delta t \tag{3.15}$$
$$\text{for } i=1,2,...,N_x,\ j=1,2,...,N_y$$

where N_x and N_y are the number of points in the $x-$ and $y-$ directions, respectively. To be clear, this would be a rectangular domain.

Equation 3.15 constitutes a set of simultaneous linear algebraic equations at each new time, t^{n+1}. It can be solved using a variety of methods, including Gauss-Seidel and Congugate Gradient, because it has favorable properties. To examine those properties, let us depart from the i,j subscript notation and, instead, introduce a single identification number for each point. Using a single number means it is not as easy to clearly specify the northern, southern, eastern, and western neighbors using the $i+1, j+1$, etc., notation. However, an indexing array could be formed that provides the single ID numbers of those neighbors, when needed. Here, we will simply denote the neighbors of point i as i_n for the neighbor to the north, i_s for the neighbor to the south, and so on. Then, Equation 3.15 becomes

$$\bar{C}\phi_{i}^{n+1}-\bar{D}\left(\phi_{i_w}^{n+1}+\phi_{i_n}^{n+1}+\phi_{i_e}^{n+1}+\phi_{i_s}^{n+1}\right)=C\phi_{i}^{n}+q\Delta t \tag{3.16}$$
$$\text{for } i=1,2,...,N$$

where $N=N_x\cdot N_y$. If we represent the system in Equation 3.16 using a matrix, we would write

$$\sum_{j=1}^{N}A_{i,j}\phi_{j}^{n+1}=C\phi_{i}^{n}+q\Delta t \qquad i=1,2,...,N \tag{3.17}$$

Discounting for now the impact of boundary conditions, the diagonal entries of A_{ij} are

$$A_{i,i} = \bar{C} \tag{3.18}$$

And, still discounting boundary conditions, each row i of A_{ij} has four additional non-zero entries, those being

$$\begin{aligned} A_{i,i_e} &= -\bar{D} \\ A_{i,i_w} &= -\bar{D} \\ A_{i,i_s} &= -\bar{D} \\ A_{i,i_n} &= -\bar{D} \end{aligned}$$

Except for the diagonal and the four columns corresponding to the ID numbers of the surrounding node IDs, the entries on row i are zero. Thus the matrix is *sparse*. Furthermore, it is *diagonally dominant*, because the sum of the off-diagonal entries' magnitudes are less than to the magnitude of the diagonal. The more diagnoally dominant a matrix is, the easier it is to solve for linear solvers; the more it resembles the identity matrix. Also, although not obvious with the above notation, the matrix is *symmetric*. To see that, consider each of the four off-diagonal terms above, for example, A_{i,i_e}. Consider the corresponding entry in the transpose of $A_{i,j}$, which is $A_{i_e,i}$. That corresponding entry is also equal to $-D$; the difference equation, Equation 3.16, when written out for node i_e would have as its western neighbor, our original node i, and its coefficient would be $\bar{D}$. Thus, the roles that point i and its eastern neighbor i_e play are reversed in the equation for that eastern neighbor.

A symmeteric, positive-definite matrix is often referred to as an "SPD" matrix. In discretization methods, these SPD matrices are also often sparse. We will discuss the storage of sparse matrices in the chapter on finite elements, and a separate chapter will be provided on iterative matrix solvers.

3.1.3 Crank-Nicholson Time Integration

It is easier to describe more sophisticated time marching schemes if we move to vector notation. In so doing, like in the previous section, we must depart from $i-j$ numbering of the finite difference points and, instead, use a "natural" point numbering, which for our current application could be simply

$$p = i + (j-1)N_x \tag{3.19}$$

where N_x is the number of points in the $x-$ direction. The north neighbor is $p+N_x$, the sound neighbor is $p-N_x$, the eastern neighbor is $p+1$ and the western neighbor is $p-1$. The unknown *phi* values are $\phi_i, i = 1,2,3,...,N$ where $N = N_xN_y$. The second-order approximation to the second spatial derivative can be represented as a matrix where, on each row, there is a $-4D/h^2$ on the diagonal, D/h^2s in the columns representing the four neighbors, and zeros elsewhere. Then, our *semi-* discrete system is

$$C\frac{\partial \phi}{\partial t} + \mathbf{A}\phi = \mathbf{q} \tag{3.20}$$

where ϕ is a vector of length N containing the ϕ_i unknown values.

We introduce a scalar value θ, which can vary between zero and 1, i.e., $0 \geq \theta \leq 1$, and use that to select where, temporally, ϕ is evaluated in the spatial derivative. Specifically, we write

$$C\frac{\phi^{n+1} - \phi^n}{\Delta t} + \mathbf{A}\left(\theta\phi^n + (1-\theta)\phi^{n+1}\right) = \mathbf{q} \tag{3.21}$$

When $\theta = 0$, we recover the Forward Euler method. When $\theta = 1$, we recover the Backward Euler method. When $\theta = 1/2$, we have the Crank-Nicholson method. Let us rearrange the equation above, to write

$$\left(C\mathbf{I} + \Delta t(1-\theta)\mathbf{A}\right)\phi^{n+1} = -\theta\Delta t\mathbf{A}\phi^n + C\phi^n + \mathbf{q} \tag{3.22}$$

The Crank-Nicholson method is second-order in time and is unconditionally stable for the diffusion equation.

3.2 Finite Difference Method for Electromagnetic Fields

Starting in the early 2000s, the term “mimetic” was introduced into discretization methods. The meaning of that term has evolved over the years, but in general, it has come to refer to the idea of forming a discretization of a differential operator that “mimics” properties of other differential operators. For example, later we will develop a mimetic method for the diffusion equation that mimics the property of the Gauss Divergence Theorem. Success at developing such an operator produces a symmetric, positive definite matrix regardless of how irregular is the grid to which the operator is applied. Other investigators have develop discretizations of the Navier-Stokes momentum and specific internal energy equations that automatically satisfy total energy conservation; this method is often referred to as the “compatable hydro” method.

Back in 1966 a finite difference discretization of the third and fourth equations in Maxwell's Equations (Ampere's Law and Faraday's Law) was developed by K. Yee that automatically satisfied the first two equations. Before the term "mimetic" had been introduced, Yee's noteworthy achievement became the standard discretization of the EM equations. First, we recap the governing equations in PDE form:

$$\nabla \cdot \mathbf{E} = Q/\varepsilon \tag{3.23}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{3.24}$$

$$\mu\varepsilon \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{B} \tag{3.25}$$

$$\frac{d\mathbf{B}}{dt} = -\nabla \times \mathbf{E} \tag{3.26}$$

For sake of clarity, it is helpful to write out some of the cross products just as a reminder:

$$\nabla \times \mathbf{B} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ B_x & B_y & B_z \end{vmatrix} \tag{3.27}$$

$$= \left(\frac{\partial B_z}{\partial y} - \frac{\partial B_y}{\partial z}\right)\mathbf{i} - \left(\frac{\partial B_z}{\partial x} - \frac{\partial B_x}{\partial z}\right)\mathbf{j} + \left(\frac{\partial B_y}{\partial x} - \frac{\partial B_x}{\partial y}\right)\mathbf{k} \tag{3.28}$$

Consider a grid that is uniform in the $x-y-z$ directions, with uniform spacing h in all three directions. Yee's discretization uses a staggered grid approach where the values of **E** are stored at the vertices and the values of **B** are stored at the cell centers. An $i-j-k$ logical vertex numbering system is introduced so that the position of any vertex is

$$\text{Position of vertex } i,j,k = (ih, jh, kh) \tag{3.29}$$

The cells are also identified using an $i-j-k$ index, specifically, a cell's ID is the same ID as its vertex with the lowest $i-j-k$ index. So the position of each cell center is offset from that "bottom, lower, left" vertex by $h/2$:

$$\text{Position of the center of cell } i,j,k = \left(ih + \frac{h}{2}, jh + \frac{h}{2}, kh + \frac{h}{2}\right) \tag{3.30}$$

The value of the electric field at vertex $i-j-k$ at time t_n is

$$\mathbf{E}^n_{ijk} = E_x|^n_{i,j,k}\mathbf{i} + E_x|^n_{i,j,k}\mathbf{j} + E_x|^n_{i,j,k}\mathbf{k} \tag{3.31}$$

and the value of the magnetic field at cell $i-j-k$ at time t_n is

$$\mathbf{H}^n_{ijk} = H_x|^n_{i,j,k}\mathbf{i} + H_x|^n_{i,j,k}\mathbf{j} + H_x|^n_{i,j,k}\mathbf{k} \tag{3.32}$$

The cross product of **E** is needed at the cell center in order to advance **H** in time. Likewise, the cross product of **B** is needed at the vertex to advance **E** in time. Yee's discretization, explicit in time, is as follows:

$$\frac{E_x|^{n+1/2}_{i,j,k} - E_x|^{n-1/2}_{i,j,k}}{\Delta t} = \frac{1}{\mu\varepsilon}\left(\frac{B_z|^n_{i,j,k} - B_z|^n_{i,j-1,k}}{\Delta y} - \frac{B_y|^n_{i,j,k} - B_y|^n_{i,j,k-1}}{\Delta z}\right) \tag{3.33}$$

for E_x, and for the other two directions it is similar:

$$\frac{E_y|^{n+1/2}_{i,j,k} - E_y|^{n-1/2}_{i,j,k}}{\Delta t} = -\frac{1}{\mu\varepsilon}\left(\frac{B_z|^n_{i,j,k} - B_z|^n_{i-1,j,k}}{\Delta x} - \frac{B_x|^n_{i,j,k} - B_x|^n_{i,j,k-1}}{\Delta z}\right) \tag{3.34}$$

and

$$\frac{E_z|^{n+1/2}_{i,j,k} - E_z|^{n-1/2}_{i,j,k}}{\Delta t} = \frac{1}{\mu\varepsilon}\left(\frac{B_y|^n_{i,j,k} - B_y|^n_{i-1,j,k}}{\Delta x} - \frac{B_x|^n_{i,j,k} - B_x|^n_{i,j-1,k}}{\Delta y}\right) \tag{3.35}$$

Analogous equations are used for the magnetic field:

$$\frac{B_x|^{n+1}_{i,j,k} - B_x|^n_{i,j,k}}{\Delta t} = \frac{1}{\varepsilon}\left(\frac{E_z|^{n+1/2}_{i,j,k} - E_z|^{n+1/2}_{i,j-1,k}}{\Delta y} - \frac{E_y|^{n+1/2}_{i,j,k} - E_y|^{n+1/2}_{i,j,k-1}}{\Delta z}\right) \tag{3.36}$$

$$\frac{B_y|^{n+1/2}_{i,j,k} - B_y|^n_{i,j,k}}{\Delta t} = -\frac{1}{\varepsilon}\left(\frac{E_z|^{n+1/2}_{i,j,k} - E_z|^{n+1/2}_{i-1,j,k}}{\Delta x} - \frac{E_x|^{n+1/2}_{i,j,k} - E_x|^{n+1/2}_{i,j,k-1}}{\Delta z}\right) \tag{3.37}$$

and

$$\frac{B_z|^{n+1/1}_{i,j,k} - B_z|^n_{i,j,k}}{\Delta t} = \frac{1}{\varepsilon}\left(\frac{E_y|^{n+1/2}_{i,j,k} - E_y|^{n+1/2}_{i-1,j,k}}{\Delta x} - \frac{E_x|^{n+1/2}_{i,j,k} - E_x|^{n+1/2}_{i,j-1,k}}{\Delta y}\right) \tag{3.38}$$

Acknowledging here that the article, "Some Numerical Techniques for Maxwells Equations in Different Types of Geometries,", by Bengt Fornberg, CU-Boulder was very helpful in preparing this section.

Chapter 4

Finite Element Discretization

4.1 Galerkin Method

To fix ideas, we consider the Galerkin finite element method applied to the steady-state convection-diffusion equation [2]. The classical statement of the mathematical problem is

$$-\frac{\partial}{\partial x_k}\left(k\frac{\partial c}{\partial x_k}\right)=q \tag{4.1}$$

subject to various types of boundary conditions, e.g.,

$$\begin{aligned} c(x_k) &= c_b(x_k) \qquad \text{fixed BCs} \\ \frac{\partial c}{\partial n} &= f(x_k,t)\text{fixed gradient} \end{aligned}$$

We form the weighted residuals statement of the problem by introducing the concept of the residual, which is simply the point-wise difference between the left-hand side and right-hand side, i.e., we move the source term on the right hand side to the left hand side, and define the result as the residual, $r(x_k,t)$,

$$r(x_k,t)\equiv-\frac{\partial}{\partial x_k}\left(k\frac{\partial c}{\partial x_k}\right)-q \tag{4.2}$$

Clearly, we seek $c(x_k)$ such that $r(x_k)$ is zero everywhere. Next, we introduce the concept of a linear vector space of functions from which we desire to construct an approximate solution. At this point, we will not yet specify what that linear vector space is, but one could imagine examples like sines and cosines with wavelengths that are increasing in integer multiples, or perhaps some group of polynomials that span a space.

We will denote the members of the vector space as $w_i(x_k), i = 1,2,3,...,N$, where N is the number of functions in that space. Likewise, we will introduce the inner product between two function $f(x_k)$ and $g(x_k)$ in that space to be the integral over their product over the domain of the problem

$$\langle f(x_k), g(x_k) \rangle = \int_{\Omega} f(x_k) g(x_k) dV \tag{4.3}$$

Next we form what is sometimes called the "weighted residual" statement of the problem. The weighted residual statement is that we seek an approximate solution $c^h(x_k)$ such that

$$\langle r_i(x_k), w_i(x_k) \rangle = 0 \text{ for } i = 1,2,3,....,N \tag{4.4}$$

or, written out as an integral, we seek c^h such that

$$\int_{\Omega} \left[-\frac{\partial}{\partial x_k} \left(k \frac{\partial c}{\partial x_k} \right) - q \right] w_i(x_k) dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.5}$$

Notice the $w_i(x_k)$ at the end of the integrand.

It is worth imagining, for a moment, that instead of a finite number of functions w_i, we could make the above statement true for all possible functions $w_i, i = 1,2,3,....$ That type of thought experiment is helpful in that it provides a different perspective on the value of the weighted residual statement. Figure 4.1 shows a residual due to some choice of function c^h, as in one chosen arbirarily or randomly. One can see that if w_i can be *anything*, then such an arbitrarily chosen function would eventually fail the test, i.e., fail to satisfy Equation 1. So, in a sense, the w functions are testing the proposed solution c_h. For this reason, the w_i are often called "test functions".

But returning to the inner product idea, we can also see that the weighted residual statement is projecting the residual against each member of the vector space, and in that sense, we are attempting to minimize the residual with respect to that vector space.

There are other important observations to be made about the weighted residual statement. First, like the classical statement, it requires that the solution c be second-order differentiable, that not only is its slope continuous, but the derivative of its slope is continuous. This mathematical requirement means that the classical statement of the problem cannot be valid over regions where changes in material

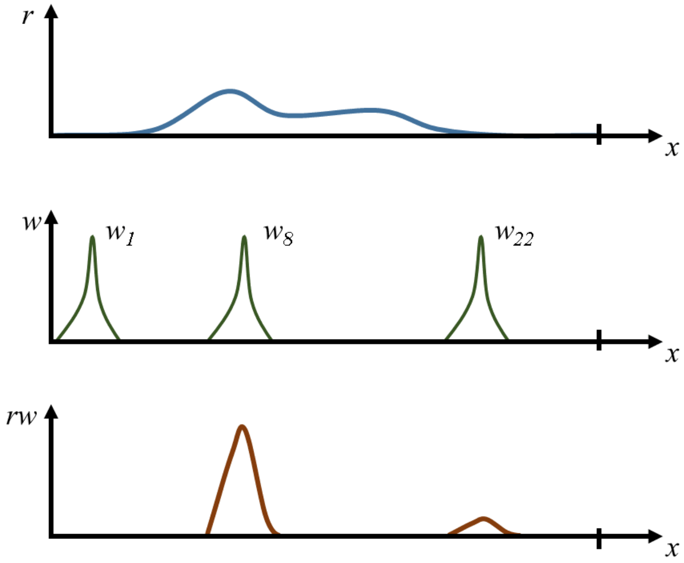


Figure 4.1: Illustration of the role test functions play in the weighted residuals statement of the problem.

properties, sources, or boundary conditions cause that requirement to be violated. What must be done in those cases is that the classical statement must be broken up by region, where new conditions are introduced along those new boundaries.

This aspect of the classical and weighted residual statements provides motivation for making a change in the statement of the problem that decreases the continuity requirements on c. We can apply the Gauss Divergence Theorem on the term involving the second spatial derivative on c and, in so doing, convert that into a first spatial derivative and a boundary term. To do that, we start with an integral that has w_i inside the gradient operator, then we use the product rule to separate them,

$$\int_\Omega \nabla \cdot (k\nabla c w_i) dV = \int_\Omega k\nabla c \cdot \nabla w_i dV + \int_\Omega \nabla \cdot (k\nabla c) w_i dV \tag{4.6}$$

Next we apply the Gauss Divergence Theorem to the left-hand side

$$\int_S k\nabla c \cdot \mathbf{n} w_i dS = \int_\Omega k\nabla c \cdot \nabla w_i dV + \int_\Omega \nabla \cdot (k\nabla c) w_i dV \tag{4.7}$$

Solving for the last term on the right-hand side,

$$\int_\Omega \nabla \cdot (k\nabla c) w_i dV = -\int_\Omega k\nabla c \cdot \nabla w_i dV + \int_S k\nabla c \cdot \mathbf{n} w_i dS \tag{4.8}$$

Writing the above result in the notation we have been using in this chapter, and swapping signs for easier use,

$$-\int_{\Omega} \frac{\partial}{\partial x_k}\left(k\frac{\partial c}{\partial x_k}\right) w_i dV = \int_{\Omega} k\frac{\partial c}{\partial x_k}\frac{\partial w_i}{\partial x_k} dV - \int_S k\frac{\partial c}{\partial x_k} n_k w_i dS \tag{4.9}$$

Using that result in our weighted residual statement produces

$$\int_{\Omega} k\frac{\partial c}{\partial x_k}\frac{\partial w_i}{\partial x_k} dV - \int_S k\frac{\partial c}{\partial x_k} n_k w_i dS - \int_{\Omega} q w_i dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.10}$$

or

$$\int_{\Omega} k\frac{\partial c}{\partial x_k}\frac{\partial w_i}{\partial x_k} dV = \int_S k\frac{\partial c}{\partial x_k} n_k w_i dS + \int_{\Omega} q w_i dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.11}$$

which is referred to as the “weak statement”, where the word “weak” refers to the fact that the continuity requirements on c are weaker than the classical and weighted residual statements. The convection (second) term in the integrand notwithstanding, notice how the statement of the problem involves more symmetry to it by the shifting of one of the derivatives from the solution to the test function w_i.

To keep the mathematics simpler, we will no longer track the surface integral term; we will drop it from the development, moving forward. However, before doing so, note that it is that surface integral term that naturally accommodates a gradient boundary condition, i.e., a flux boundary condition. In practice, when the finite element method is used on a problem that has a flux boundary condition, the surface integral is retained on that part of the boundary, and the $k\nabla c$ value in the integrand is replaced with the specific gradient condition. On those parts of the boundary that have a fixed or Dirichlet condition, the integral is zero for another reason: Specifically, that the test functions are set to zero on that boundary. But for this development, it will be easier to follow with out the surface integral and so we drop it.

We have made only one approximation so far, which is that we have restricted the space of all possible functions w_i to a finite set of N functions that form a basis for a vector space. And the statement with which we are left is not yet making a statement about how we intend to approximate c.

In the Galerkin finite element method, we choose the same basis for approximating c as is used for the basis functions in the vector space that w_i spans.

Specifically, we seek coefficients $c_j(t)$ such that

$$c(x_k) = \sum_{j=1}^{N} c_j w_j(x_k) \tag{4.12}$$

Introducing the approximation into the weak statement produces the Galerkin finite element statement of the problem,

$$\sum_{j=1}^{N} \int_{\Omega} kc_j \frac{\partial w_j}{\partial x_k} \frac{\partial w_i}{\partial x_k} dV = \int_{\Omega} qw_i dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.13}$$

Because we have not yet specified the actual functions w_i, we cannot yet perform actual computations. That aside, it is worth noting that the above statement is N equations (one for each w_i) in N unknowns.

There is much to the choice of the basis functions, and one can imagine all sorts of possibilities. For smooth solutions on simply shaped domains, it may be possible to choose w_j to be a series of sines or cosines, like a Fourier series. This choice gives rise to what are known as “spectral methods”. For solutions that are not smooth, there may be no benefit to choosing higher order basis functions since they may not be able to accommodate discontinuities.

4.2 Finite Element Linear Basis Functions

In the finite element method, we divide the spatial domain into elements, and we use those elements to define the basis functions for the linear vector space.

Among the simplist of elements is the triangle. Shown in Figure 4.2 is a small mesh consisting of triangles. Each triangle element is defined by three points, sometimes called “nodes”. Let us consider one individual node in the mesh. Note that it part of multiple triangular elements. We can define planar functions on each triangle that are unity at the node we are considering and zero on the triangle’s other two nodes, the resulting function looks like the tent shown in Figure 4.3. Note that these new planar functions that we have introduced are also zero outside of their parent triangular elements.

This tent function is one of the basis functions that can be used with triangular meshes. Note that there can be a similar “tent” basis function associated with each point/node.

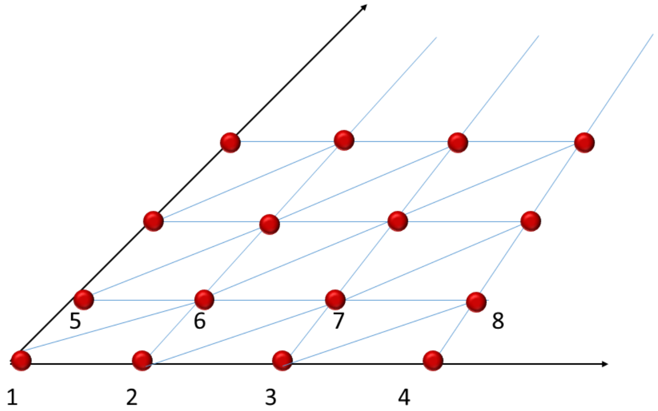


Figure 4.2: Simple triangle grid.

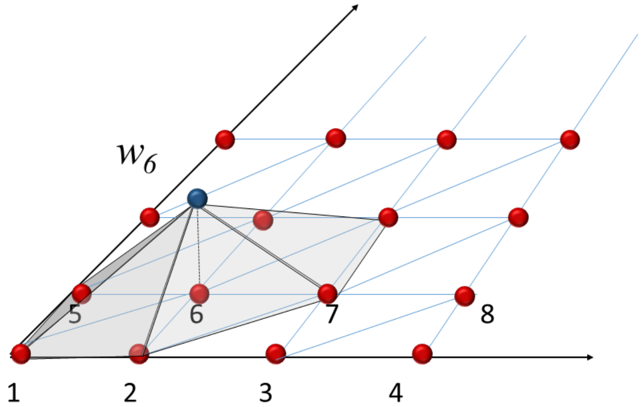


Figure 4.3: Simple triangle grid “tent” basis function.

In one dimension, it is easier to visualize how the different “tent” basis functions work together to form the finite element approximation of the solution. Recall from Equation 4.12 that we are stipulating the the finite element approximation be coefficients $c_i, i = 1, 2, ..., N$ times each of the respective basis functions. Imagining for a moment that we knew those c_i values, i.e., the values of the approximate solutioni, Figure shows how when those c_i values are multiplied by their respective basis function and then the basis functions are summed, the approximate solution is constructed througout the entire simulation domain.

Now we turn our attention to finding those c_i values.

4.3 Finite Element Statement of the Problem

We now introduce Equation 4.12 into the weak statement to produce

$$\int_\Omega \left[k \sum_{j=1}^{N} \frac{\partial c_j w_j}{\partial x_k} \frac{\partial w_i}{\partial x_k} = \int_\Omega qw \right] w_i(x_k) dV \text{ for } i = 1, 2, 3, ..., N \tag{4.14}$$

The summation and integral orders can be reversed. Also, we can move the constants c_j outside of the integrals, enabling us to write the above equation this way,

$$\sum_{j=1}^{N} c_j \int_\Omega \left[k \frac{\partial w_j}{\partial x_k} \frac{\partial w_i}{\partial x_k} \right] dV = \int_\Omega q w_i dV \text{ for } i = 1, 2, 3, ..., N \tag{4.15}$$

In the above equation, we can identify the integral as two matrices of $i - j$ entries, i.e., if we define

$$D_{ij} \equiv \int_\Omega \frac{\partial w_i}{\partial x_k} \frac{\partial w_j}{\partial x_k} dV$$

and the q term as

$$Q_i \equiv \int_\Omega q w_i dV$$

then we can write the above equation as

$$\sum_{j=1}^{N} \left(D_{ij} \right) c_j = Q_i \text{ for } i = 1, 2, 3, ..., N \tag{4.16}$$

The above statement is a system of N simultaneous algebraic equations for the N unknowns c_i and constitutes the Galerkin Finite Element statement of the diffusion problem using linear triangular elements.

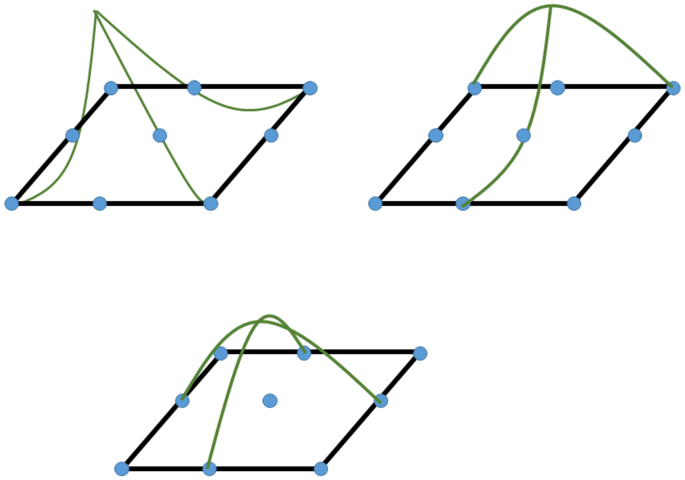


Figure 4.4: Illustration of some biquadratic shape functions components.

4.4 Higher Order Elements

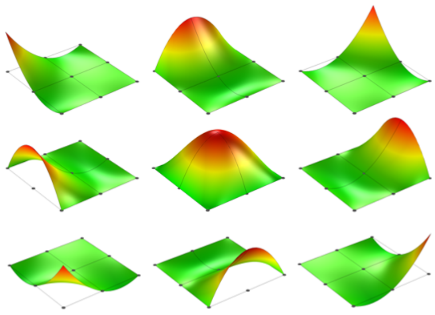
The triangular elements with linear basis functions are among the easiest to visualize and draw. However, for problems where the solution is known to be fairly smooth, they provide a less optimal approximation basis. Consider, for example, the bi-quadratic elements shown in Figure 4.4 and Figure 4.5. On each element there are nine polynomial functions, each of which is equal to unity at one of the element's nodes and zero at the others. They are bi-quadratic because they provide a quadratic basis in the $x-$ and $y-$ directions. Likewise, we can have quadratic triangular elements, as shown in Figure.

Note that for both of these quadratic elements, the basis functions at a node are formed by joining the contributions from all of the adjoining elements.

4.5 Computing the Integrals

4.5.1 Why we Break them up by Elements

Equation 4.16 seems rather straightforward until one must endeavor to compute the matrix entries, i.e., the integrals that comprise D_{ij}. For linear triangles, one



From the Comsol website.

Figure 4.5: Biquadratic shape functions. Credit: Comsol

might imagine carrying out the integration by hand. However, for the bi-quadratic basis functions the integration would be more difficult. And, we have not ever required the elements to be simple, i.e., the triangles could be rotated with respect to he axis. Furthermore, we will later see that the bi-quadratic elements can have curved edges.

For these reasons, it is very convenient to introduce a numerical method for computing the integrals in the finite element matrics. And in developing that method, it is very valuable to combine the following observations:

- We recognize that each of the integrals in Equation 4.16 are for a particular combination of basis functions. We also recall that the basis functions are only non-zero over the elements connected with their node. This means that the integral of the product of two basis functions or the product of their derivatives is zero everywhere except over the elements touching *both* of those nodes.

- Any of the particular matrix entries for a $i-j$ combination of basis functions can be computed piece by piece, specifically, the contribution of the integral from each of the elements touching both i and j can be computed separately and simply added together. To be clear, we will introduce some notation to help us. First, we introduce the notation $e(i,j)$, which will represent the list of elements that touch *both* node i and j. Likewise, we will let $\int_e$ represent the integral over element e; again, here e will be one of the members of $e(i,j)$, and we will denote e's contribution to the integral using a superscript, i.e., Then,

$$D^e_{ij} \equiv \int_e \frac{\partial w_i}{\partial x_k}\frac{\partial w_j}{\partial x_k} dV \tag{4.17}$$

Then, the global matrix entry for the $i-j$ basis function combination is the sum over the contributing elements,

$$D_{ij} = \sum_{\text{all } e \text{ in } e(i,j)} D^e_{ij} \tag{4.18}$$

Computations of the integrals are carried out with the above three facts as central motivators. It turns out there are other motivators as well, which will become apparent later. For now, suffice it to say in the finite element method we typically compute integrals on an element-by-element basis and then assemble them into what we call the "global" matrix D_{ij}.

4.5.2 Element Integrals

Since we compute the integrals on each element, it is extremely helpful to introduce a systematic way of performing the computations that is amenable to an algorithm. To that end, we introduce a way to represent each element as a set of parameters that can be used in the same algorithm regardless of the location or orientation of the element.

The Master Element

We will consider the bi-quadratic element here; other elements are treated elsewhere. We introduce the "master element", as shown in Figure 4.6 Notice that it is on a the $\hat{x}_1 - \hat{x}_2$ plane, not the $x_i, i = 1,2$ plane. It is drawn that way because it is not a "physical" element, i.e., it is not in the actual geometry of our problem at hand. Rather, it forms the foundation of a mathematical tool that we will use for the integration process.

More capability is needed for this master element to become functional, however. The first capability we add is the same basis functions that exist on the actual

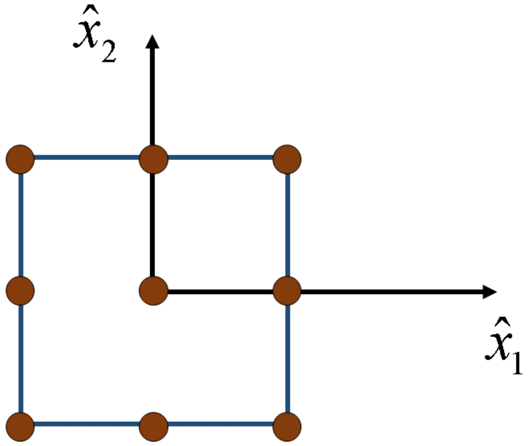


Figure 4.6: Master element for biquadratic elements.

physical elements over which we want to compute the integrals. Shown in Figure 4.5 are the nine basis functions on the master element. Note that each one is a function of $\hat{x}_1 - \hat{x}_2$; we are still developing a mathematical tool.

We can use this capability to create a mathematical connection between each point in the $\hat{x}_1 - \hat{x}_2$ plane to a corresponding point in the $x_1 - x_2$ plane. But to do that, we need some additional notation:

- Global Node Number Map: We need a map between the local node numbers in the master element, which range from 1 to 9, to the global node numbers for the particular physical element e we are considering. Clearly, that map will be different for each element. We will name that map $n[e, p_m]$. Here, $n[e, p_m]$ gives the global node or point number in element e corresponding to master element point number p_m where, again $i = 1, 2, 3, ..., 9$. We use the square brackets here on purpose, to denote that this is a data structure and not a mathematical function.

- Coordinate Array: We need a way to store the coordinates of each point. For that purpose, we introduce the notation $x_i[p]$, where as usual, $i = 1, 2$ for each coordinate direction in 2D and p is the global node number for point p. As with $n[e, p_m]$ above, the square brackets denote a data structure.

Later, we will see parentheses associated with x_i, and in those cases the parentheses represent a mathematical functional notation.

- Master Element Basis Functions: The basis functions on the master element, as shown in Figure 4.5, will be referred to as $\phi^m_{p_m}(\hat{x}_1,\hat{x}_2)$. Each of the nine element basis functions is associated with one of the master element points. Note that, except for $\phi^m_9(\hat{x}_1,\hat{x}_2)$ which is self-contained in the master element, the other basis functions only represent part of the actual basis functions on the physical mesh.

Mapping Master Element to Physical Elements

Using the above tools we can perform any operation on the physical element by performing and analogous operation on the master element while transforming the results appropriately. To do that, we first start with the ability to map any point $(\hat{x}_1,\hat{x}_2)$ to a corresponding point (x,y) inside a the element being considered. We show that mapping here and then discuss it. The mapping, from $\hat{x}_1,\hat{x}_2$ to x,y for element e is

$$x_i(\hat{x}_1,\hat{x}_2) = \sum_{p_m=1}^{9} x_i[n[e,p_m]]\phi_{p_m}(\hat{x}_1,\hat{x}_2) \tag{4.19}$$

First, as mentioned above, the parentheses after x_i denote a functional dependence on $\hat{x}_1$ and $\hat{x}_2$ whereas on the right hand side, the square brackets denote a data structure. Second, note that the above equation is true for both coordinate directions, x_1 and x_2. Third, consider the point $\hat{x}_1 = -1, \hat{x}_2 = -1$. At that point, all the master element basis functions are zero except ϕ_1, which is unity. Thus in the sum over the nine basis functions, only the first one survives and is simply

$$x_i(-1,-1) = x_i[n[e,1]] \tag{4.20}$$

The inside of the right hand side in the above equation is the data structure, $n[e,1]$, which is the global node number for one of the nodes in the physical element. Thus the right hand side references the data structure giving the x_1 and x_2 values for that node. Thus, the above expression is correct, and a similar one can be constructed for each of the points in the master element.

For the rest of the $\hat{x}_1,\hat{x}_2$ points, those not at nodes, the master element basis functions are not necessarily zero. Rather, the provide an interpolation tool between the nodal locations.

The interpolating ability of the coordinate map in Equation 4.19 can be seen clearly by traversing the set of points along the bottom of the master element, i.e.,

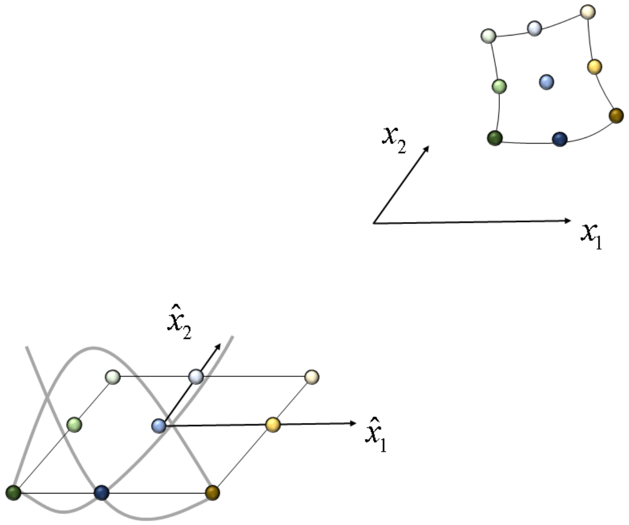


Figure 4.7: Position interpolation with a biquadratic element.

where $\hat{x}_2 = 0$, and $\hat{x}_1$ ranges between -1 and 1. Along that line segment, only three of the master element basis functions are nonzero, those being ϕ_1, ϕ_5, and ϕ_2. If element e is as shown in 4.7, where it is valuable to note that the middle point is not exactly in the middle of the other two, the function that maps $\hat{x}_1$ to x_1 is shown in Figure 4.7.

It will also be important to compute derivatives of the coordinate mapping in Equation 4.19. They are

$$\frac{\partial x_i}{\partial \hat{x}_j} = \sum_{p_m=1}^{9} x_i[n[e,p_m]] \frac{\partial \phi_{p_m}}{\partial \hat{x}_j} \tag{4.21}$$

Again, note that the above equation is true for each coordinate direction, $i = 1,2$.

Lastly, we use the master element basis functions to compute the basis functions on the physical element as a function of the master element coordinates, i.e., we can write

$$w_{n[e,p_m]}(x_1,x_2) = \phi_{p_m}(\hat{x}_1,\hat{x}_2) \tag{4.22}$$

We stipulate that the value of the basis function $w_{n[e,p_m]}$ at x_1,x_2 is is equal to the corresponding basis function $\phi_{n[e,p_m]}(\hat{x}_1,\hat{x}_2)$, on the master element, nothing this dependence: $x_1(\hat{x}_1,\hat{x}_2), x_2(\hat{x}_1,\hat{x}_2)$. In other words, we build into the method that

$$w_{n[e,p_m]}(x_1,x_2) = \phi_{p_m}(\hat{x}_1,\hat{x}_2) \tag{4.23}$$

To be more clear, we establish the following operational mechanisms for the master element as it relates to the physical element:

$$
\begin{aligned}
(x_1,x_2) &= x_1(\hat{x}_1,\hat{x}_2), x_2(\hat{x}_1,\hat{x}_2) \\
w_{n[e,p_m]}(x_1,x_2) &= \phi_{p_m}(\hat{x}_1,\hat{x}_2)
\end{aligned}
$$

Performing Computations using the Master Element

We have the ability to map any point $\hat{x}_1,\hat{x}_2$ to any point x_1,x_2 for each element e. Now we want to use that ability to compute an integral over element e using the master element. To do that, we first need to relate $dA = dx_1 dx_2$ in the physical element to $d\hat{A} = d\hat{x}_1 d\hat{x}_2$ in the master element, for that relationship quantifies how each infinitessimal contribution in the master element must be scaled to accurately represent the corresponding infinitessimal contribution in the physical element.

Now consider the infinitessimal area in the $\hat{x}_1,\hat{x}_2$ plane. It is a small square, with the lower left point at $\hat{x}_1,\hat{x}_2$ and its upper right point at $\hat{x}_1+d\hat{x}_1,\hat{x}_2+d\hat{x}_2$. Two vectors can be defined along the bottom (B) and left (L) side of the infinitessimal area, those being

$$\hat{\mathbf{r}}_B = d\hat{x}_1\hat{\mathbf{i}} \tag{4.24}$$

$$\hat{\mathbf{r}}_L = d\hat{x}_2\hat{\mathbf{j}} \tag{4.25}$$

The area of that infinitessimal is equal to the magnitude of the cross product of those vectors, i.e.,

$$d\hat{A} = |\hat{\mathbf{r}}_B \times \hat{\mathbf{r}}_L| \tag{4.26}$$

$$= \begin{vmatrix} d\hat{x}_1 & 0 \\ 0 & d\hat{x}_2 \end{vmatrix} \tag{4.27}$$

$$= d\hat{x}_1 d\hat{x}_2 \tag{4.28}$$

Now consider the mapping of that infinitessimal area into the physical domain. It may not map to a square, or even a rectangle. Rather, it may be rotated and each side may have a length that is very different from each other and from that of the master element's infinitessimal area. The lower left corner of the original infinitessimal maps to the coordinate

$$x_1(\hat{x}_1,\hat{x}_2) \quad , \quad x_2(\hat{x}_1,\hat{x}_2) \tag{4.29}$$

or simply

$$x_1,x_2 \tag{4.30}$$

for easier visualization; however we must not lose track of the functional dependence. The lower right corner maps to the coordindate

$$x_1 + \frac{\partial x_1}{\partial \hat{x}_1} d\hat{x}_1 \quad , \quad x_2 + \frac{\partial x_2}{\partial \hat{x}_1} d\hat{x}_1 \tag{4.31}$$

Note that only the $d\hat{x}_1$ term is needed in the above Taylor series expansions because, in the master element, $d\hat{x}_2 = 0$ as we traverse the bottom edge of the infinitessimal area. Likewise, the upper left corner of the infinitessimal area maps to the coordinate

$$x_1 + \frac{\partial x_1}{\partial \hat{x}_2} d\hat{x}_2 \quad , \quad x_2 + \frac{\partial x_2}{\partial \hat{x}_2} d\hat{x}_2 \tag{4.32}$$

With three of the four corners of the area in the physical space now known, we can form the vectors along two sides that correspond to $\hat{\mathbf{r}}_R$ and $\hat{\mathbf{r}}_L$. They are

$$\begin{aligned}
\mathbf{r}_B &= \frac{\partial x_1}{\partial \hat{x}_1} d\hat{x}_1 \mathbf{i} + \frac{\partial x_2}{\partial \hat{x}_1} d\hat{x}_1 \mathbf{j} \\
\mathbf{r}_L &= \frac{\partial x_1}{\partial \hat{x}_2} d\hat{x}_2 \mathbf{i} + \frac{\partial x_2}{\partial \hat{x}_2} d\hat{x}_2 \mathbf{j}
\end{aligned}$$

Analogous to the area of the infinitessimal in the master element, the area in the physical space is

$$\begin{aligned}
dA &= |\mathbf{r}_B \times \mathbf{r}_L| \qquad\qquad (4.33) \\
&= \begin{vmatrix} \frac{\partial x_1}{\partial \hat{x}_1} d\hat{x}_1 & \frac{\partial x_2}{\partial \hat{x}_1} d\hat{x}_1 \\ \frac{\partial x_1}{\partial \hat{x}_2} d\hat{x}_2 & \frac{\partial x_2}{\partial \hat{x}_2} d\hat{x}_2 \end{vmatrix} \\
&= \left(\frac{\partial x_1}{\partial \hat{x}_1} \frac{\partial x_2}{\partial \hat{x}_2} - \frac{\partial x_2}{\partial \hat{x}_1} \frac{\partial x_1}{\partial \hat{x}_2} \right) d\hat{x}_1 d\hat{x}_2 \\
&= \left(\frac{\partial x_1}{\partial \hat{x}_1} \frac{\partial x_2}{\partial \hat{x}_2} - \frac{\partial x_2}{\partial \hat{x}_1} \frac{\partial x_1}{\partial \hat{x}_2} \right) d\hat{A}
\end{aligned}$$

Note that the derivatives in the parenthesis are computed by multiplying the derivatives of the master element basis functions relative to the master element coordinates by the corresponding coordinate values and summing them all together, i.e., Equation 4.21.

The coefficient in the parentheses is a scaling factor for an infinitessimal area in the master element plane to the corresponding mapped area in the physical plane. It is a term that is important in other ways, as well. For other purposes that shall become evident later, we define what is called the "Jacobian" matrix as

$$\mathbf{J} \equiv \begin{bmatrix} \frac{\partial x_1}{\partial \hat{x}_1} & \frac{\partial x_2}{\partial \hat{x}_1} \\ \frac{\partial x_1}{\partial \hat{x}_2} & \frac{\partial x_2}{\partial \hat{x}_2} \end{bmatrix} \tag{4.34}$$

Then the expression above may be re-written using $\mathbf{J}$ as

$$|\mathbf{J}| d\hat{A} = dA \tag{4.35}$$

Therefore, if we wish to integrate a function $f(x_1,x_2)$ over the physical element, we may instead integrate it over the master element, i.e.,

$$\int_e f(x_1,x_2) dx_1 dx_2 = \int_{\hat{e}} f(\hat{x}_1,\hat{x}_2) |\mathbf{J}| d\hat{x}_1 d\hat{x}_2 \tag{4.36}$$

Because the master element is square, it is much easier to develop algorithms for performing the numerical integration. The determinant of the Jacobian is, in general, not a constant. That fact precludes the ability to do an analytical evaluation of the integral.

Integration of $f(x_1,x_2)$'s derivatives, i.e., the evaluation of

$$\int_e \left(\frac{\partial f}{\partial x_1} \right) dA \tag{4.37}$$

or integrals with combinations of derivatives require additional consideration. Specifically, recall from the derivation of the finite element statement of the problem, Equation 4.17, that we must evaluate

$$D^e_{ij} \equiv \int_e \frac{\partial w_i}{\partial x_k} \frac{\partial w_j}{\partial x_k} dV \tag{4.38}$$

If we are to perform this integration on the master element, we must not only transform dA using the determinant of the Jacobian, we must transform the derivatives as well.

We start in that process by recalling that x_i can be expressed as a function of $\hat{x}_1, \hat{x}_2$ via the mapping in Equation 4.19. Via the chain rule:

$$dx_1 = \frac{\partial x_1}{\partial \hat{x}_1} d\hat{x}_1 + \frac{\partial x_1}{\partial \hat{x}_2} d\hat{x}_2$$
$$dx_2 = \frac{\partial x_2}{\partial \hat{x}_1} d\hat{x}_1 + \frac{\partial x_2}{\partial \hat{x}_2} d\hat{x}_2$$

The above expressions may be written in matrix form as

$$\begin{bmatrix} dx_1 \\ dx_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial \hat{x}_1} & \frac{\partial x_1}{\partial \hat{x}_2} \\ \frac{\partial x_2}{\partial \hat{x}_1} & \frac{\partial x_2}{\partial \hat{x}_2} \end{bmatrix} \begin{bmatrix} d\hat{x}_1 \\ d\hat{x}_2 \end{bmatrix} \tag{4.39}$$

Notice that the matrix the Jacobian, which was derived above for the infinitessimal area mapping. So

$$\begin{bmatrix} dx_1 \\ dx_2 \end{bmatrix} = \mathbf{J} \begin{bmatrix} d\hat{x}_1 \\ d\hat{x}_2 \end{bmatrix} \tag{4.40}$$

We can invert the above 2×2 system using Cramer's Rule to get

$$\begin{bmatrix} d\hat{x}_1 \\ d\hat{x}_2 \end{bmatrix} = \frac{1}{|\mathbf{J}|} \begin{bmatrix} \frac{\partial x_2}{\partial \hat{x}_2} & -\frac{\partial x_1}{\partial \hat{x}_2} \\ -\frac{\partial x_2}{\partial \hat{x}_1} & \frac{\partial x_1}{\partial \hat{x}_1} \end{bmatrix} \begin{bmatrix} dx_1 \\ dx_2 \end{bmatrix} \tag{4.41}$$

Now we will endeavor to develop another type of expression for $d\hat{x}_1$ and $d\hat{x}_2$, which will be combined with the one above in order to find a way to transform the spatial derivates to the master element. That new expression relies on the concept of an inverse map, one that maps the physical point x, y to a point $\hat{x}_1, \hat{x}_2$ in the master element. Although we have not determined that inverse map, one exists in principle, as long as $|\mathbf{J}| \neq 0$. So for now, under the assertion the following inverse maps exists, we write

$$\hat{x}_1 = \hat{x}_1(x_1, x_2)$$
$$\hat{x}_2 = \hat{x}_2(x_1, x_2)$$

and from that use the chain rule to write

$$d\hat{x}_1 = \frac{\partial \hat{x}_1}{\partial x_1} dx_1 + \frac{\partial \hat{x}_1}{\partial x_2} dx_2$$
$$d\hat{x}_2 = \frac{\partial \hat{x}_2}{\partial x_1} dx_1 + \frac{\partial \hat{x}_2}{\partial x_2} dx_2$$

In matrix form, the above equations are

$$\begin{bmatrix} d\hat{x}_1 \\ d\hat{x}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial \hat{x}_1}{\partial x_1} & \frac{\partial \hat{x}_1}{\partial x_2} \\ \frac{\partial \hat{x}_2}{\partial x_1} & \frac{\partial \hat{x}_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} dx_1 \\ dx_2 \end{bmatrix} \tag{4.42}$$

So, now we have two expressions, in Equations 4.39 and 4.42, for $d\hat{x}_1$ and $d\hat{x}_2$, which means the matrices in those equations are equal, i.e.,

$$\begin{bmatrix} \frac{\partial \hat{x}_1}{\partial x_1} & \frac{\partial \hat{x}_1}{\partial x_2} \\ \frac{\partial \hat{x}_2}{\partial x_1} & \frac{\partial \hat{x}_2}{\partial x_2} \end{bmatrix} = \frac{1}{|\mathbf{J}|} \begin{bmatrix} \frac{\partial x_2}{\partial \hat{x}_2} & -\frac{\partial x_1}{\partial \hat{x}_2} \\ -\frac{\partial x_2}{\partial \hat{x}_1} & \frac{\partial x_1}{\partial \hat{x}_1} \end{bmatrix} \tag{4.43}$$

It is instructive to write out these equalities explicity. The equations below are the same as the matrix equality above. They only look different because the summations over the master element basis functions derivates, originally appearing in Equation 4.21 are shown.

$$\frac{\partial \hat{x}_1}{\partial x_1} = \frac{1}{|\mathbf{J}|}\frac{\partial x_2}{\partial \hat{x}_2} = \frac{1}{|\mathbf{J}|}\sum_{p_m=1}^{9} x_2[n[e,p_m]]\frac{\partial \phi_{p_m}}{\partial \hat{x}_2} \tag{4.44}$$

$$\frac{\partial \hat{x}_1}{\partial x_2} = -\frac{1}{|\mathbf{J}|}\frac{\partial x_1}{\partial \hat{x}_2} = -\frac{1}{|\mathbf{J}|}\sum_{p_m=1}^{9} x_1[n[e,p_m]]\frac{\partial \phi_{p_m}}{\partial \hat{x}_2} \tag{4.45}$$

$$\frac{\partial \hat{x}_2}{\partial x_1} = -\frac{1}{|\mathbf{J}|}\frac{\partial x_2}{\partial \hat{x}_1} = -\frac{1}{|\mathbf{J}|}\sum_{p_m=1}^{9} x_2[n[e,p_m]]\frac{\partial \phi_{p_m}}{\partial \hat{x}_1} \tag{4.46}$$

$$\frac{\partial \hat{x}_2}{\partial x_2} = \frac{1}{|\mathbf{J}|}\frac{\partial x_1}{\partial \hat{x}_1} = \frac{1}{|\mathbf{J}|}\sum_{p_m=1}^{9} x_1[n[e,p_m]]\frac{\partial \phi_{p_m}}{\partial \hat{x}_1} \tag{4.47}$$

One important thing to recognize in the above four equations is that they are readily computable. Each term in the summation is a known coordinate value of a physical point/node times the spatial derivative of the master element basis function relative to its native coordinate system $\hat{x}_1, \hat{x}_2$.

With these derivatives now available to us, it is a straightfoward matter to transform the spatial derivatives of the basis functions on the physical element to spatial derivatives on the master element where we carry out the integration. Using the chain rule, we write

$$\frac{\partial w_{n[e,p_m]}}{\partial x_1} = \frac{\partial \phi_{p_m}}{\partial \hat{x}_1}\frac{\partial \hat{x}_1}{\partial x_1} + \frac{\partial \phi_{p_m}}{\partial \hat{x}_2}\frac{\partial \hat{x}_2}{\partial x_1}$$

$$\frac{\partial w_{n[e,p_m]}}{\partial x_2} = \frac{\partial \phi_{p_m}}{\partial \hat{x}_1}\frac{\partial \hat{x}_1}{\partial x_2} + \frac{\partial \phi_{p_m}}{\partial \hat{x}_2}\frac{\partial \hat{x}_2}{\partial x_2}$$

Again, every term on the right hand side is readily computable. The first terms in each product on the right hand side is the derivative of the basis function relative to its native variables on the master element. In other words, they are the straightforward derivatives of polynomials functions. The second terms in each product on the right hand side are slightly more involved, but are straightforward to compute. They are the terms in Equations 4.44-4.47.

Numerical Integration on the Master Element

There are multiple ways to perform numerical integration of functions on the master element. One particular method that has found broad use in the finite element method is Gauss Quadrature. The two dimensional integral is approximated using a summation of samples multiplied by weights. In other words,

$$\int_{-1}^{1}\int_{-1}^{1} F(\hat{x}_1,\hat{x}_2)d\hat{x}_1 d\hat{x}_2 = \sum_{m=1}^{N_m} W_m F(\hat{x}_{1m},\hat{x}_{2m}) \tag{4.48}$$

where W_m are predefined weights and the array of points $(\hat{x}_{1m},\hat{x}_{2m})$.

Element Assembly into the Global Matrix

We have covered all of the basic components in setting up the Galerkin finite element method for the convection diffusion problem. It involves the computation of element matrices D^e_{ij} and the summation of those matrices to form the element matrix. It is helpful to contemplate how that assembly is accomplished. First, note that the matrix is sparse. Each row in the matrix is associated with a node/point in the mesh. Each column in a particular row represents the relationship between that node and the other nodes surrounding it. Because the basis functions are non-zero only on the elements touching that node, the columns for nodes that are in elements not touching that node are zero.

Because the matrix is sparse, it is much more efficient computationally to store only the non-zero entries. Here we describe one of the more straightforward data structures. We define three arrays, as follows:

- `Alen[i]` This integer array gives the number of non-zero columns in row `i` of the matrix.

- `Jcoef(i,j)` This integer array stores the column numbers that are non-zero in row `i` of the matrix; note that `i` ranges from one to `Alen(i)`.

- `Acoef(i,j)` This floating point array stores the non-zero values in row `i`.

To summarize, consider a full, square matrix stored in a two-dimensional array: `A[i][j]`, where `i` corresponds to row i in A_{ij} and `j` corresponds to row j. The sparse storage format is such that `Acoef[i,j] = A[i][Jcoef[i,j]]`.

4.6 Transient Finite Element Discretization

We consider the Galerkin finite element method applied to the transient convection-diffusion equation. The classical statement of the mathematical problem is

$$\frac{\partial c}{\partial t} - \frac{\partial}{\partial x_k}\left(k\frac{\partial}{\partial x_k}\right) = q \tag{4.49}$$

subject to initial conditions

$$c(x_k,t) = c_k(x_k) \tag{4.50}$$

and various types of boundary conditions, e.g.,

$$\begin{aligned} c(x_k) &= c_b(x_k,t) \qquad \text{fixed BCs} \\ \frac{\partial c}{\partial n} &= f(x_k,t) \text{ fixed gradient} \end{aligned}$$

The weighted residual statement is that we seek the approximate solution $c^h(x_k,t)$ such that

$$\int_\Omega \left[\frac{\partial c}{\partial t} - \frac{\partial}{\partial x_k}\left(k\frac{\partial}{\partial x_k}\right) - q\right] w_i(x_k)dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.51}$$

Application of the Gauss Divergence Theorem, and dismissing the surface integral term for simplicity, yields the weak statement

$$\int_\Omega \left[\frac{\partial c}{\partial t}w_i + k\frac{\partial c}{\partial x_k}\frac{\partial w_i}{\partial x_k} - qw\right] w_i(x_k)dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.52}$$

As an extension of the Galerkin method applied to the steady state problem, here we seek coefficients that are functions of *time*, i.e., $c_j(t)$, such that

$$c(x_k,t) = \sum_{j=1}^{N} c_j(t)w_j(x_k) \tag{4.53}$$

Notice here that the coefficients are functions of time and the basis functions are not. Introducing the basis function approximation into the weak statement produces the Galerkin finite element statement of the problem,

$$\sum_{j=1}^{N}\int_\Omega \left[\frac{\partial c_j(t)}{\partial t}w_i + kc_j(t)\frac{\partial w_j}{\partial x_k}\frac{\partial w_i}{\partial x_k} - qw_i\right] dV = 0 \text{ for } i = 1,2,3,....,N \tag{4.54}$$

Chapter 5

Control Volume Method Applications

5.1 Compatible Hydro

In this section, a Lagrangian finite volume discretization for the conservation of momentum in gas dyanmics, Equation 2.6 is described. Being "Lagrangian" means the grid will move with the material. Therefore, conservation of mass is automatically satisfied. Also, a way to evolve the specific internal energy that is compatable with our momentum discretization will be described.

To begin, we define an unstructured mesh consisting of points, p and cells c as shown in Figure 5.1. We define the following:

$$\sum_p = \text{ The sum of all points touching cell } c$$

$$\sum_c = \text{ The sum of all cells touching point } p$$

Next, we define a staggered grid, as shown in Figure. The control volume for the momentum equation is centered on the points. Meanwhile, we evolve specific internal energy in the cells. Thus we make use of the following subscripts:

$$\begin{aligned} \mathbf{u}_p &= \text{ velocity of point } p \\ \mathbf{x}_p &= \text{ position of point } p \\ \rho_c &= \text{ density in cell } c \\ e_c &= \text{ specific internal energy in cell } c \\ P_c &= \text{ pressure in cell } c \end{aligned}$$

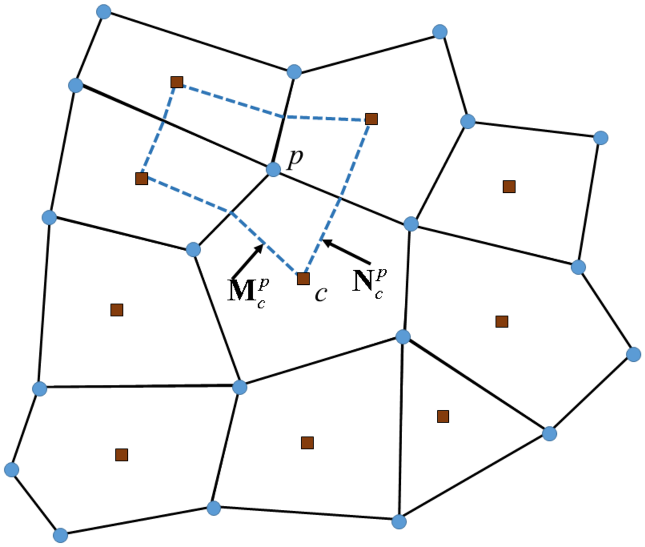


Figure 5.1: The staggered grid mesh for compatible hydro. The control volume for momentum is centered around the vertex, with a portion of each touching cell being part of it. Specific internal energy is stored in the cell.

Note that pressure is given by an EOS, $P_c = P(\rho_c, e_c)$. The algorithm proceeds in three steps:

1. The momentum is updated one-half of a time step by integrating the pressures of the surrounding cells, times the appropriate surface normals, around the momentum control volume:

$$\mathbf{u}_p^{n+1/2} = \mathbf{u}_p^n + \frac{\Delta t}{2m_p} \sum_c p_c \left(\mathbf{N}_c^p + \mathbf{M}_c^p\right) \tag{5.1}$$

   where m_p the mass associated with point p. After that, the position of the vertices are updated a half time-step:

$$\mathbf{x}_p^{n+1/2} = \mathbf{x}_p^n + \frac{\Delta t}{2} \mathbf{u}_p^{n+1/2} \tag{5.2}$$

   which enables us to use the new shape of the cells to update that density, $\rho_c^{n+1/2}$, at each cell. Recall mass is constant so as the grid distorts density changes in each cell.

2. The energy is updated at the half-time step by accounting for the work done by each cell on the points it accelerated in Step 1.

$$e_c^{n+1/2} = e_c^n - \frac{\Delta t}{2m_c} \sum_p p_c \left(\mathbf{N}_c^p + \mathbf{M}_c^p\right) \cdot \mathbf{u}_p \tag{5.3}$$

   We use this energy at the half time-step along with the density $\rho_c^{n+1/2}$ to compute a mid-step pressure in each cell, $P_c^{n+1/2}$.

3. We leapfrog over the half-timestep, starting back at the beginning of the time step but using the mid-step values:

$$\begin{aligned}
\mathbf{u}_p^{n+1} &= \mathbf{u}_p^n + (\Delta t/m_p) \sum_c p_c^{n+1/2} \left(\mathbf{N}_c^p + \mathbf{M}_c^p\right) \\
\mathbf{x}_p^{n+1} &= \mathbf{x}_p^n + \Delta t \mathbf{u}_p^{n+1/2} \\
e_c^{n+1} &= e_c^n - (\Delta t/m_c) \sum_p p_c^{n+1/2} \left(\mathbf{N}_c^p + \mathbf{M}_c^p\right) \cdot \mathbf{u}_p^{n+1/2}
\end{aligned}$$

There are multiple issues with the algorithm as stated. Most notably, there is no dissipation. The physics associated with a shock, which can be studied via the Rankine-Hugoniot equations are not embodied in the algorithm. Oscillations, even to the point of causing the simulation to fail completely, result. Dissipation can be added via artificial viscosity. To each cell's pressure, a q_c is added where q_c is the rate of compression. The von Neumann-Richtmeyer artificial viscosity is an effective and popular formulation and is

$$q_{vNR} = \rho_o C \left[l_c \frac{1}{V} \frac{\Delta V}{\Delta t} \right]^2 \tag{5.4}$$

where C is a constant, typically about 1.2, V is the current cell volume, and ρ_o is the cell's initial density.

5.2 Mimetic Method for Diffusion

The derivations of many discretization methods start with the governing equations themselves. But this mimetic method method starts, instead, with the discretization of the following integral relationship

$$\int_e \mathbf{H} \cdot \nabla \phi dV = -\int_e \phi \nabla \cdot \mathbf{H} dV + \int_{\partial e} \phi \mathbf{H} \cdot \mathbf{n} dA \tag{5.5}$$

This relationship, which is true for any arbitrary, sufficiently smooth functions ϕ and $\mathbf{H}$ on a cell e in a mesh, is a result of the product rule for differentiation and the Gauss Divergence theorem. It is chosen as the starting point because it demonstrates very desirable properties of the continuous differential operators, properties that are also desired for the discrete operators. For example, the above relationship can be used to demonstrate that the divergence and gradient operators are adjoint to each other, or similarly, that the Laplacian operator is self-adjoint and positive definite. Therefore, by discretizing this relationship, the same properties can be enforced in the discrete operators.

To see the connection of the Gauss-Green formula to the diffusion problem, consider this simple form of the diffusion equation

$$\nabla \cdot (-\nabla \phi) = q \tag{5.6}$$

on the domain V with zero Dirichlet boundary conditions. The equation may be written using the two operators defined below:

$$D \equiv \nabla \cdot \qquad G \equiv -\nabla \tag{5.7}$$

Using these operators, the simple diffusion equation is written as follows:

$$D[G(\phi)] = q \tag{5.8}$$

To discretize this equation, the continuous solution ϕ is exchanged for a discrete solution. Likewise, the continuous operators are replaced with discrete operators, which are matrices of numbers. These numbers are chosen such that when multiplied by adjacent discrete ϕ values, they mimic the continuous operators (derivatives). The discrete form is therefore

$$\mathbf{DG}\phi = \mathbf{q} \tag{5.9}$$

From matrix algebra we know that if $\mathbf{D}$ is the adjoint [1] of $\mathbf{G}$, then their product is a symmetric-positive-definite (SPD) matrix. It is possible to choose $\mathbf{D}$ and $\mathbf{G}$ that mimic the derivatives they represent, but are not adjoint. In fact, that is traditionally the approach: Develop discrete forms of the derivatives by starting with the partial differential equation, and then deal with whatever matrix results.

But to develop a linear system that is SPD by design, one must explicitly enforce the desire that $\mathbf{D}$ and $\mathbf{G}$ are adjoint. The way that desire is enforced

[1] For real matrices, the word transpose can be used instead of adjoint.

is motivated by the fact that the continuous operators D and G are themselves adjoint. To show this fact, start with the inner product of $G\phi$ with the arbitrary function H, apply the product rule, and then the Gauss Divergence Theorem:

$$\begin{aligned}
\langle G\phi, H\rangle_1 &= \int_V -\nabla\phi \cdot \mathbf{H} dV && (5.10)\\
&= \int_V -\nabla \cdot (\mathbf{H}\phi)\, dV + \int_V \phi \nabla \cdot \mathbf{H} dV \qquad \text{(Product rule)}\\
&= \int_{\partial v} \phi \mathbf{H} \cdot \mathbf{n} dV + \int_V \phi \nabla \cdot \mathbf{H} dV \qquad \text{(Gauss Divergence Theorem)}\\
&= \int_V \phi \nabla \cdot \mathbf{H} dV \qquad \text{(Zero boundary condition)}\\
&= \int_V \nabla \cdot \mathbf{H} dV\\
&= \langle \phi, DH\rangle_2
\end{aligned}$$

The above set of simple steps demonstrates that derivatives associated with G may be transferred from ϕ to H through integration by parts. In so doing, the two inner products defined using G and D are shown to be equal. In a more intuitive sense, one can see by looking at the results of the above operations that there is kind of “symmetry” to G and D. In particular, they play similar roles in their two spaces of functions. To summarize, then, it is the Gauss-Green formula that embodies the fact that D and G are adjoint to each other. By discretizing it, that adjointness is preserved in the discrete D and G operators. [2]

To begin the process, we write the diffusion equation as two first order equations:

$$\nabla \cdot \mathbf{F} = q \qquad (5.11)$$

$$\mathbf{F} = -\nabla\phi \qquad (5.12)$$

Equation 5.12 is the constitutive model, with a coefficient set to unity for simplicity. We insert it directly Equation 5.5 to get

$$\int_e \mathbf{H} \cdot \mathbf{F} dV = -\int_e \phi \nabla \cdot \mathbf{H} dV + \int_{\partial e} \phi \mathbf{H} \cdot \mathbf{n} dA \qquad (5.13)$$

We will discetize this to derive a discrete **D** and **G** that are adjoint to each other.

[2] The version resulting from Equation 5.10 is simplified because of the zero Dirichlet boundary conditions. In general, the boundary terms are retained as in Equation 5.5.

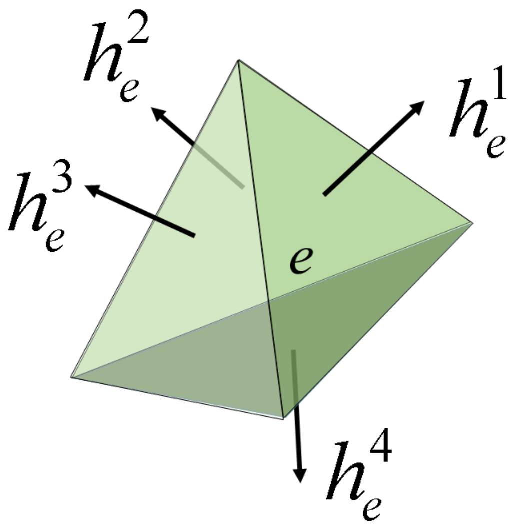


Figure 5.2: Face normal vectors for **H**. Similar vectors are constructed for **F**.

5.2.1 Discretization of the Gauss-Green Formula for Tetrahedra

The two terms on the right-hand side of Equation 5.13 are discretized under the following constructs; see also Figure 5.2.

- We represent the vector field **H** on the tetrahedron using four face-normal values, h_e^1, h_e^2, h_e^3, and h_e^4, each one being normal to a side of the tetrahedron "tet".

- We define fluxes f_e^k to be face normal, just like the h_e^k.

- We define A_e^k to be the areas of those sides.

- We define a cell-averaged value of the solution, ϕ_e^c

- We define face-centered values of the solution, $\phi_e^k, k = 1, 2, 3, 4$.

With those definitions in mind, we form these compact representations:

$$\mathbf{D}_e = \begin{bmatrix} A_e^1 & & & \\ & A_e^2 & & \\ & & A_e^3 & \\ & & & A_e^4 \end{bmatrix} \qquad \mathbf{h}_e = \begin{bmatrix} h_e^1 \\ h_e^2 \\ h_e^3 \\ h_e^4 \end{bmatrix} \tag{5.14}$$

and use them to discretize the two integrals on the right-hand side of Equation 5.13:

$$\int_e \phi \nabla \cdot \mathbf{H} dV \approx \phi^c \sum_{k=1}^{4} h^k A^k = \phi^c \mathbf{D}_e \mathbf{h}_e \tag{5.15}$$

$$\int_{\partial e} \phi \mathbf{H} \cdot \mathbf{n} dA \approx \sum_{k=1}^{4} h^k A^k \phi^k = \mathbf{h}_e^T \mathbf{D}_e \boldsymbol{\phi}_e \tag{5.16}$$

It is no coincidence that we chose the letter D for $\mathbf{D}_e$. Note that it is actually our discrete divergence operator. Recall the intention of this process is to develop a discrete $\mathbf{D}$ and $\mathbf{G}$ operator pair that are adjoint. Here, we have chosen the $\mathbf{D}_e$ on the element. From this point forward, we shall be seeking $\mathbf{G}_e$.

Now, to discretize the left-hand side, we approximate its integral as the sum of four sub-volume integrals, where each sub-volume is associated with a corner of the tetrahedron, i.e., a vertex of the tetrahedron. Because we have a face-based approach, we need to somehow combine the face vectors for $\mathbf{h}_e$ and $\mathbf{f}_e$ into single vectors $\mathbf{H}_e^k$ and $\mathbf{F}_e^k$ for that sub-volume. To be precise, we seek to combine the three h_e^k and f_e^k vectors from the three k faces touching vertex k into a single vector that represents those values of $\mathbf{H}_e^k$ and $\mathbf{F}_e^k$ at the vertex. And as a reminder, we actually only need their dot product.

Each vertex k is adjacent to three sides, which for now will be labeled a, b, and c. The vector $\mathbf{F})e^k$ can be expressed using either a Cartesian basis or the f^a, f^b, and f^c basis:

1. In Cartesian basis: $\mathbf{F}_e^k = \left(f_x^k, f_y^k, f_z^k\right)$ or
2. In the face-normal basis: $\mathbf{F}_e^k = \left(f^a, f^b, f^c\right)^k$, i.e., the adjacent faces' normal vectors.

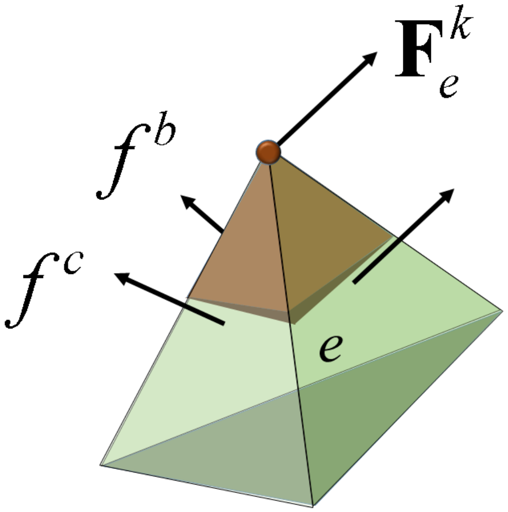


Figure 5.3: Face normal vectors for **H**. Similar vectors are constructed for **F**.

The same is true for $\mathbf{H}_e^k$. The (unknown) Cartesian basis is related to to the (known) $\left(f^a, f^b, f^c\right)^k$ basis as follows:

$$\begin{bmatrix} f^a \\ f^b \\ f^c \end{bmatrix} = \begin{bmatrix} n_x^a & n_y^a & n_z^a \\ n_x^b & n_y^b & n_z^b \\ n_x^c & n_y^c & n_z^c \end{bmatrix} \begin{bmatrix} f_x^k \\ f_y^k \\ f_z^k \end{bmatrix} \tag{5.17}$$

where, the matrix contains the unit normal vectors on the three adjacent faces, again here labeled a, b, and c. For example, n_x^a is the $x-$component of the normal to side a. In compact notation,

$$\mathbf{f}_{abc}^k = \mathbf{N}_k \mathbf{F}_e^k \tag{5.18}$$

from which it follows that

$$\mathbf{F}_e^k = \mathbf{N}_k^{-1} \mathbf{f}_{abc}^k \tag{5.19}$$

The usual dot product between $\mathbf{F}_e^k$ and $\mathbf{H}_e^k$ is sought as follows:

$$\mathbf{F}_e^k \cdot \mathbf{H}_e^k = (\mathbf{N}_k)^{-1} \mathbf{f}_{abc}^k \cdot (\mathbf{N}_k)^{-1} \mathbf{h}_{abc}^k = \left[(\mathbf{N}_k)^{-1}\right]^T (\mathbf{N}_k)^{-1} \mathbf{f}_{abc}^k \cdot \mathbf{h}_{abc}^k \tag{5.20}$$

These are small, easily manipulated matrices. We define

$$\bar{\mathbf{S}}_e^k \equiv \left[(\mathbf{N}_k)^{-1}\right] \mathbf{N}^{-1} \tag{5.21}$$

so that

$$\mathbf{F}_e^k \cdot \mathbf{H}_e^k = \mathbf{S}_e^k \mathbf{f}_{abc}^k \mathbf{h}_{abc}^k \tag{5.22}$$

The above results are for a particular vertex, k, and its three adjoining sides, a, b, and c. Hence the vectors in Equation 5.23 are of length three and $\bar{\mathbf{S}}_e^k$ is 3×3.

To express the dot product using the vector of all face-normals on the tetrahedron, a new 4×4 matrix $\mathbf{S}_e$ is formed from $\bar{\mathbf{S}}^k$ containing the same values but with zeros interspersed so that the face not adjacent to k is ignored. With that, we could write

$$\mathbf{F}_e^k \cdot \mathbf{H}_e^k = \mathbf{S}_e^k \mathbf{f}_e \mathbf{h}_e \tag{5.23}$$

And that is used in the summation for the volume integral on the left-hand side of Equation 5.13.

$$\int_e \mathbf{H} \cdot \mathbf{F} dV \approx \frac{1}{4} V_e \sum_{k=1}^{4} \mathbf{S}_e^k \cdot \mathbf{H}_e \cdot \mathbf{F}_e^k \tag{5.24}$$

Each of the $\mathbf{S}_e^k$ in the above summation could be summed together to represent a single $\mathbf{S}_e$ matrix for the tetrahedron. At the same time, the factor of $V_e/4$ can be absorbed into $\mathbf{S}_e$ to write

$$\int_e D^{-1} \mathbf{H} \cdot \mathbf{F} dV \approx \mathbf{S}_e \cdot \mathbf{h}_e \cdot \mathbf{f}_e \tag{5.25}$$

Putting the approximations for the left-hand side integral and the two integrals on the right side, we have, for element e,

$$\mathbf{S}_e \mathbf{h}_e \cdot \mathbf{f}_e = \phi^c \mathbf{D}_e \mathbf{h}_e + \mathbf{D}_e \mathbf{h}_e \boldsymbol{\phi}_e \tag{5.26}$$

or

$$\mathbf{S}_e \mathbf{h}_e \cdot \mathbf{f}_e = \phi^c \mathbf{D}_e \mathbf{h}_e \left(\mathbf{I} \phi^c + \boldsymbol{\phi}_e \right) \tag{5.27}$$

The above equation is a discrete version of the Gauss-Green formula, which is the relationship we seek to mimic in our diffusion discretization method. It contains the discrete divergence operator $\mathbf{D}_e$ that we have chosen, and it involves the cell-centered as well as face-centered solution values ϕ_c and $\boldsymbol{\phi}_e$. We have made no statement about $\mathbf{h}_e$, however. We developed our discretization of the Gauss-Green formula for arbitrary $\mathbf{h}_e$. Now, we will exercise that fact by selecting “test values” of for $\mathbf{h}_e$.

5.2.2 Finding the Adjoint

We choose four different $\mathbf{h}_e$ vectors, and substitute those vectors, one at a time, into Equation 5.27. Each time we substitute one of our chosen test $\mathbf{h}_e$ vectors, we create a scalar equation that involves our four flux values, i.e., the components of $\mathbf{f}_e$ and the ϕ values. The test vectors for $\mathbf{h}_e$ that we choose are

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \quad \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \quad \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \text{ and } \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{5.28}$$

Again, each one of the above choices, when substituted into Equation 5.27 results in a scalar equation involving all four unknown ϕ_e, the unknown ϕ^c values, and the four unknown flux values. Those four equations can be written in matrix form as

$$\mathbf{M}_e\mathbf{f}_e = \mathbf{B}_e\boldsymbol{\phi}_e^+ \tag{5.29}$$

where $\boldsymbol{\phi}_e^+$ is $\boldsymbol{\phi}_e$ appended with ϕ_c. The above system can be inverted,

$$\mathbf{f}_e = \mathbf{M}_e^{-1}\mathbf{B}_e\boldsymbol{\phi}_e^+ \tag{5.30}$$

Then defining

$$\mathbf{G}_e \equiv \mathbf{M}_e^{-1}\mathbf{B}_e \tag{5.31}$$

we have

$$\mathbf{f} = \mathbf{G}_e\boldsymbol{\phi}_e^+ \tag{5.32}$$

We chose that name $\mathbf{G}_e$ on purpose – note how $\mathbf{G}_e$ relates the $\boldsymbol{\phi}_e$ and ϕ_c values to the face fluxes. This is precisely the purpose of the gradient operator; $\mathbf{G}_e$ is our discrete gradient operator, derived from the Gauss-Green formula with our chosen $\mathbf{D}_e$.

5.2.3 Application to Transient Diffusion

We now consider the diffusion equation written as two first-order equations as follows:

$$c\frac{\partial \phi}{\partial t} + \nabla \cdot \mathbf{F} = q \tag{5.33}$$

$$\mathbf{F} = -\nabla\phi \tag{5.34}$$

The constitutive equation, Equation 5.34, has already been embodied in the previous sections' analysis (refer back to Equation 5.13). Therefore, attention now focuses on Equation 5.33, the conservation equation. Each term in Equation 5.33 is now integrated over the volume of the zone, which is V_e.

$$c\frac{\partial}{\partial t}\left(\frac{1}{V}\int_V \phi dV\right) + \left(\frac{1}{V}\int_V \nabla \cdot \mathbf{F} dV\right) = \frac{1}{V}\int_V q dV \tag{5.35}$$

The volume-average for the intensity and right-hand side are defined as follows:

$$\phi^c \equiv \frac{1}{V}\int_V dV \qquad \bar{q} \equiv \frac{1}{V}\int_V q dV \tag{5.36}$$

In keeping with the discretization of the Gauss-Green formula, and specifically, Equation 5.14, the following approximation is made for the divergence:

$$\left(\frac{1}{V_e}\int_V \nabla \cdot \mathbf{F} dV\right) \approx \frac{1}{V_e}\sum_{i=1}^{s} A^i f^i \tag{5.37}$$

Note that this approximation must be chosen for the flux-intensity relationship (i.e., discrete gradient) in Equation 5.14 to be relevant. Specifically, Equation 5.32 is the result of discretizing the Gauss-Green formula using the discretization for the divergence operator shown in Equation 5.14 (and 5.37).

Now, using Equations 5.36 and 5.37 in Equation 5.35 and multiplying by volume, the semi-discrete form is produced:

$$Vc\frac{\partial \phi^c}{\partial t} + \sum_{i=1}^{s} A^i f^i = V\bar{q} \tag{5.38}$$

The following definitions are made

$$C \equiv Vc \quad \text{and} \quad \bar{Q} \equiv V\bar{q} \tag{5.39}$$

so that Equation 5.38 may be written as

$$C\frac{\partial \phi^c}{\partial t} + \sum_{i=1}^{s} A^i f^i = \bar{Q} \tag{5.40}$$

The fully discrete form is obtained by using Backwards-Euler time differencing. Let ϕ^c denote the cells volume-averaged intensity at the previous time step. Then, the approximation to Equation 5.40 becomes

$$C\frac{\phi^c - \hat{\phi}^c}{\Delta t} + \sum_{i=1}^{s} A^i f^i = \bar{Q} \tag{5.41}$$

Rearranging terms produces

$$\left(\frac{C}{\Delta t}\right)\phi^c + \sum_{i=1}^{s} A^i f^i = \bar{Q} + \left(\frac{C}{\Delta t}\right)\hat{\phi}^c \tag{5.42}$$

The development up to this point has focused on individual cells. But the solution, ϕ, is expected to be continuous, which means that if side k of zone z is coincident with side k' of zone z' , then $\phi_z^k = \phi_{z'}^{k'}$. Also, the flux must be continuous, which following the same notation means that $f_z^k = -f_{z'}^{k'}$ Those two ideas along with Equation 5.41 allow for the formation of a symmetric system, regardless of the shape of the tetrahedra in the mesh.

Chapter 6

C++ Programming

6.1 Motivation

All technical documents take time to define terms. They do so as their explanations unfold, defining terms along they way, or at the beginning in a table, or a combination of both. The definition of variables and terms are, in a sense, the "language" of the technical document. We seek to communicate ideas in this course thorough programming, and so we must take time to define that language as well. This chapter is meant to provide the bare minimum number of defitions in the C++ programming language to allow for the effective communication of ideas for this course. This chapter is certainly not meant to be an exhaustive description of C++. However, it is hoped that enough material is provided here to enable the student not currently familiar with C++ to engage with the course material and, perhaps, gain some long-term usefulness from the language.

6.2 Basic Structure

A bare-minimum C++ program looks like this:

```
void main()
{
}
```

It does not do anything, but it will execute when compiled. The first word is `void` and is mean to indicate that the name of the function (main) does not return any values. Sometimes, Unix-based systems require all executable to return something that can be used for detecting errors. So, on a Unix system it may be necessary to modify the program like this:

```
int main()
{
return 0;
}
```

This program also will not do anything, but it will return the value of zero to the calling program, which in this case would be the operating system.

The `main` program above can call other functions either ones that we write, ones provided in a library, or ones pre-written and stored as part of the C++ release. Here, we provide a function for `main` to call through this new version

```
int SayHello()
{
return 0;
}

int main()
{
int err;
err = SayHello();
return 0;
}
```

In this new version, of our code, the SayHello routine is defined in the file first so that the compiler already knows about it when it encounters the call to SayHello, in the line that reads `err = SayHello()`. There are ways to tell the compiler about SayHello without putting it first in the file. We can, for example, tell the compiler what the SayHello routine looks like, without giving the full code for it. Eventually, of course, the SayHello must be provided. Below, the “form” of SayHello is provided at the top so the compiler knows to expect in in the main routine, and its actual definition is include later.

```
int SayHello();

int main()
{
int err;
err = SayHello();
return 0;
}

int SayHello()
{
return 0;
}
```

That first line of this version, the one that states `int SayHello();` is called the “prototype” for the SayHello routine.

Usually there are numerous prototypes, and so they may be put in a separate file, often called a “header” file, that can be included. In following that idea, our code will now consist of two files, one of being the header file. So now, to aide our discussion, we will begin using “code blocks” that provide the filename in gray and the code in blue. Using a header file, our code now looks like this:

main.cpp

```
#include "SayHello.h"

int main()
{
  int err;
  err = SayHello();
  return 0;
}

int SayHello()
{

}
```

where

SayHello.h

```
int SayHello();
```

Analgous to a mathematical text, where one might say $x = 4 \cdot y$ where $y = 13$, the code blocks above define main.cpp to be the first code block in blue and SayHello.h to be second code block in blue.

There is much more that can and will occur with header files. For example, header files can actually contain the code itself. So we could have this as our code:

main.cpp

```
#include "SayHello.h"

int main()
{
  int err;
  err = SayHello();
  return 0;
}
```

where

SayHello.h

```
int SayHello()
{


}
```

Here, the entire code for SayHello has, in a sense, returned to the top of main.cpp, but is doing so through an "include" command.

6.3 Printing to the Screen

The subject of input and output ("I/O") is quite large; here we give information required for understanding codes in this class. We will use the C++ "iostream" capability for printing to the screen. To use it, we must include a header file in our code so the compiler has the routines we will be using, just like in our SayHello example above. In this next example, main.cpp is unchanged:

main.cpp

```
#include "SayHello.h"

int main()
{
  int err;
  err = SayHello();
  return 0;
}
```

But now SayHello.h has a print statement in it, with the required inclusion of the "iostream" capability:

SayHello.h

```
#include <iostream>

using std :: cout;
using std :: endl;

int SayHello()
{
  cout << "I am saying hello. " << endl;
};
```

This version of the code actually produces output, so we show the output here, in a green box:

TTY Output

```
I am saying hello.
```

6.4 Declaring Variables

To place values in memory so that we can work with them, we declare variables. There are multiple kinds of variables that we can declare. Two that we will use a great deal are floating point numbers and integers. It is important that we tell the compiler what type of variable we wish to declare. The reason why is because, ultimately, all variables are eventually] stored as ones and zeros in memory. It is the variable type that determines how those ones and zeros are interpeted. For example, suppose we declare the variable `i` to be an integer and store in `i`'s memory location the value 132. If we then reference that memory location later, in a different part of our code, but in so doing tell the compiler it is a floating point number (i.e., a real number with a decimal), the code will not interpret it as 132.00. Rather, it will be some very different floating point number, or perhaps not a number at all. The code has to be told in very explicit terms what type of variable is stored in a memory location.

In this example, main.cpp is unchanged from above

SayHello.h

```
#include "SayHello.h"

int main()
{
  int err;
  err = SayHello();
  return 0;
}
```

but SayHello.h declares some variables, stores values in them, and prints them to the sccren:

SayHello.h

```
#include <iostream>

using std :: cout;
using std :: endl;

int SayHello()
{
  cout << "I am saying hello. " << endl;

  float x;
  float y;

  x = 13.45;
  y = -21.5;

  cout << "I have values to share with you: " << endl;
  cout << "  (o) x = " << x                   << endl;
  cout << "  (o) y = " << y                   << endl;
};
```

Here is the output:

TTY Output

```
I am saying hello.
I have values to share with you:
  (o) x = 13.45
  (o) y = -21.5
```

6.5 Standard Template Library

There is a technology that comes with C++ called “templates”. We will not be developing any templates in this course. But for reference, a template is a routine that can accept one or more variables whose types can vary, depending on the calling routine and, in response to that, perform different actions depending on that type. For example, a template that adds to variables might do simple addition if it is passed integers. But if it is passed two strings, it might concatenate them.

Again, however, the development of templates is beyond the scope of this course. That stated, we will be making use of the “Standard Template Library”, essentially as “users”, without delving into them very deeply. Two variable types that are provided by the Standard Template Library, also called “STL” are variable sized arrays and strings.

In this example, main.cpp is unchanged from above

SayHello.h

```
#include "SayHello.h"

int main()
{
  int err;
  err = SayHello();
  return 0;
}
```

but SayHello.h declares a arraya using the standard template library, tells the compiler how big the array should be, stores values in the array, and prints them:

SayHello.h

```
#include <iostream>
#include <string>
#include <vector>

using std :: cout;
using std :: endl;
using std :: string;
using std :: vector;

#define VD  vector<double>

int SayHello()
{
  cout << "I am saying hello. " << endl;
  cout << "I use the Standard Template Library. " << endl;

  vector<double> x_array;

  x_array.resize(10);

  for ( int i = 1 ; i < 10 ; ++i )
    {
      cout << "In array location " << i   <<
              " I am storing "     << i*i << endl;
      x_array[i] = i*i;
    }


  cout << "I have values to share with you: " << endl;

  for ( int i = 1 ; i < 10 ; ++i )
    {
      cout << "x_array[" << i << "] = " << x_array[i] << endl;
    }

};
```

Here is the output:

TTY Output

```
I am saying hello.
I use the Standard Template Library.
In array location 1 I am storing 1
In array location 2 I am storing 4
In array location 3 I am storing 9
In array location 4 I am storing 16
In array location 5 I am storing 25
In array location 6 I am storing 36
In array location 7 I am storing 49
In array location 8 I am storing 64
In array location 9 I am storing 81
I have values to share with you:
x_array[1] = 1
x_array[2] = 4
x_array[3] = 9
x_array[4] = 16
x_array[5] = 25
x_array[6] = 36
x_array[7] = 49
x_array[8] = 64
x_array[9] = 81
```

6.6 Structs

The variable types discussed above are relatively simple. First we showed an example that declared two floating point values, `x` and `y` . Each represented a small place in memory in which floating point numbers, specifically single precision numbers with decimal places, were stored. Then, we showed an example that used a single letter, `x` to represent a set of 10 values in memory, in seqential order, each located using an index in brackets, e.g., `x[3]` .

This next topic continues that path of generalization. Again, to review: First we declared a single variable. Second we declared a set of variables references by a single letter and an index. Now we invent our own type of variable that, inside of it, can store different types. An analogy would be like this:

1. Declare a single variable – that's like a apartment.

2. Declare an array – that's like an apartment build where every apartment is identical.

3. Invent a struct and delcare it – that's like a big hotel, with different types of rooms, ballrooms, and a lobby. In other words, it has many types of rooms, all under one roof.

In this first example, we return to a single file that contains all we need. Here it is:

main.cpp

```
#include <iostream>

using std :: cout;
using std :: endl;

struct Circle
{
public:

  float radius;
  float xc;
  float yc;
};

int main()
{
  Circle C;

  C.radius = 1.;
  C.xc = 0.;
  C.yc = 0.;

  cout << "Here is data from our circle:"          << endl;
  cout << "   (o) Radius: " << C.radius            << endl;
  cout << "   (o) Center: " << C.xc << " " << C.yc << endl;
}
```

In the code above, we define a new type of variable, this one being called a "Circle". The key point here is to understand that we are given the privilege of defining our own variable type. While C++ provides us with integers, floats, and double precision floats, and the standard template library helps us define strings and arrays, C++ also gives us the ability to define our own custom types.

It introduces a new level of abstraction to think of defining a new type of variable. But if one remembers that a variable is simply a location or set of locations in memory referred to by a name, e.g., `x` , then perhaps it is not unreasonable to collect various types of data together under one name.

However, defining the new variable type is not enough. We must also tell the compiler that we want to reserve a space in memory to store a variable of that type. Consider this: The inventors of C++ provide a type "float". But we have to use that type in order for us to define a variable, i.e., we must have a line like this

```
float x;
```

in our code. Once we have that, we can use `x` . The same is true for a struct. We define the variable type `Circle` with the `struct Circle..` command and what follows it. But we do not reserve memory to store a variable of that type until we include a line such as

```
Circle C;
```

in our code.

The output for our code is shown here:

TTY Output

```
Here is data from our circle:
    (o) Radius: 1
    (o) Center: 0 0
```

6.7 Classes

In this section, we continue on our path of abstraction with respect to inventing new types of variables. This next abstraction adds to structs something that goes beyond only storing data in memory. It greatly expands on that idea by adding to the data storage *functionality*. This is a new concept, and seems at first to be quite a jump. Here is how it fits into our analogy:

1. Declare a single variable – that's like a apartment.

2. Declare an array – that's like an apartment build where every apartment is identical.

3. Invent a struct and declare it – that's like a big hotel, with different types of rooms, ballrooms, and a lobby. In other words, it has many types of rooms, all under one roof.

4. Invent a class and declare it – that's like a big hotel, with different types of rooms, ballrooms, and a lobby but also, now, with a front desk that will greet you, cleaning services, event planning services, etc. In other words, it not only has many types of rooms, all under one roof, it has many services as well.

To illustrate this concept, consider our next example is almost identical to the previous one, except for one change. We move the print statements inside the Circle class.

main.cpp

```
#include <iostream>

using std :: cout;
using std :: endl;

class Circle
{
public:

  float radius;
  float xc;
  float yc;

  void PrintInfo()
  {
    cout << "I am a circle.  Here is my information:" << endl;
    cout << "   (o) Radius: " << radius              << endl;
    cout << "   (o) Center: " << xc << " " << yc     << endl;
  }
};

int main()
{
  Circle C;

  C.radius = 1.;
  C.xc = 0.;
  C.yc = 0.;

  C.PrintInfo();
}
```

The output for our code is shown here:

TTY Output

```
I am a circle.  Here is my information:
   (o) Radius: 1
   (o) Center: 0 0
```

Inside a class, it is possible to create a special function that is automatically executed each time a version of the class is placed into memory. That function is called a “Constructor”. It is also possible to create another special function that is aujtomatically executed each time the version of the class is removed from meory. That function is called a “Destructor”. Both of these functions have a specific and special form. Both their names must be the same name as the class, except the destructor’s name is preceded with the ˜’ symbol. And, although neither function returns a value, they are not preceded by the word `void`. In this next example, main.cpp has been slightly modified so that the Circle class has a constructor and a destructor. Also, just for illustrative purposes, the main program has been modified so that two circles are instantiated, one is called C1 and the other is called C2.

main.cpp

```
#include <iostream>

using std :: cout;
using std :: endl;

class Circle
{
public:

  Circle()
  {
    radius = 1.;
    xc = 0.0;
    yc = 0.0;
    cout << "(o) I am a circle that just now was placed " << endl;
    cout << "    into memory (i.e., instantiated)." << endl;
  }

  ~Circle()
  {
    cout << "(o) This Circle is being removed from memory" << endl;
  }


  float radius;
  float xc;
  float yc;

  void PrintInfo()
  {
    cout << "(o) I am a circle.  Here is my information:" << endl;
    cout << "    (o) Radius: " << radius              << endl;
    cout << "    (o) Center: " << xc << " " << yc     << endl;
  }
};

int main()
{
  Circle C1;
  Circle C2;
  C1.PrintInfo();
}
```

The output for our code is shown here:

TTY Output

```
(o) I am a circle that just now was placed
    into memory (i.e., instantiated).
(o) I am a circle that just now was placed
    into memory (i.e., instantiated).
(o) I am a circle.  Here is my information:
    (o) Radius: 1
    (o) Center: 0 0
(o) This Circle is being removed from memory
(o) This Circle is being removed from memory
```

Because we instantiate into memory two copies of the Circle class, with one memory location labeled C1 and the other labeled C2, the constructor is executed twice, once for each copy. Then, in the main body of the code, we only call C1's PrintInfo() routine, not C2's. Thus, only C1 prints its radius and center. Then, as the code finishes execution and the operating system is removing all variables from memory, both C1 and C2 have their destructors called.

6.8 Macros

We will make use of macros in this class because their use makes our codes look much cleaner and simpler. We will define our macros at the top of the files, using statements that look something like this example:

```
#define iLOOP for ( int i = 1 ; i <= npoints ; ++i )
```

Then, we can make use of this macro by replacing code that would look like this:

```
for ( int i = 1 ; i <= npoints ; ++i ) Temperature[i] = IC_Temp;
```

with this

```
iLOOP Temperature[i] = IC_Temp;
```

When the compiler encounters the `iLOOP` in the above line of code, it replaces `iLOOP` with the standard for loop spelled out in our macro definition. Note that there is nothing special about the choice of letters we used in the macro, i.e., `iLOOP` was an arbitrary choice. We could have named the macro `LOOPoveri` .

We will define several macros like the one above. One disadvantage of macros is they mostly have to be memorized, which can be a hurdle when learning someone else's code. Fortunately, we will develop them one at a time and so assimilation of these definitions should be doable.

6.9 Multi-Dimensional Arrays

In this section, we go expand on our previous example where we declared an array with a single index and reserved memory for multiple entries. At the same time, we being a new code sequence; we are leaving the Circle class from previous examples behind and are embracing the Grid class.

In this first example of the series, we declare a "2D" array, i.e., an array with two indices. The code is shown below:

main.cpp

```
#include <iostream>
#include <string>

using std :: cout;
using std :: endl;
using std :: string;

class Grid
{
public:

  Grid(string _name)
  {
    name = _name;
    cout << "(o) Grid named " << name << " is going into memory." << endl;
  }

  ~Grid()
  {
    cout << "(o) Grid " << name << " is being removed from memory" << endl;
  }

  string name;
  int   nx;
  int   ny;

  void PrintInfo()
  {
    cout << "(o) Grid Information:                   " << endl;
    cout << "----------------------------------------" << endl;
    cout << "    (*) Name        : " << name           << endl;
    cout << "    (*) Num x points: " << nx             << endl;
    cout << "    (*) Num y points: " << ny             << endl;
  }
};

int main()
{
  Grid MyGrid("Square");
  MyGrid.nx = 10;
  MyGrid.ny = 10;
  MyGrid.PrintInfo();
}
```

We see in the above code a few new things: The use of the STL string datatype, a constructor that can receive an argument, and two 2-D arrays, `px` and `py`. The constructor receives a string from the main routine, which is instantiating it into memory. The constructor uses that string to set its own internal variable, `name`.

Another important new feture of this code is the introduction of 2-D arrays, `px` and `py`, which will store the $x-$ and $y-$ locations of each point in the mesh. The 2-D arrays are actually just “1-D arrays of 1-D arrays,” or in STL terminology, “vectors of vectors.” To allocate the memory for the 2-D arrays, therefore, first the 1-D arrays are alloocated with the `resize` command. Note that each element of those 1-D arays is of type `vector<double>` . Thus, each element has a resize command that can be executed, which we do to gain the second index of the 2-D arrays.

The output of the code is shown below:

TTY Output

```
(o) Grid named Square is going into memory.
(o) Grid Information:
----------------------------------------
    (*) Name        : Square
    (*) Num x points: 10
    (*) Num y points: 10
(o) Grid Square is being removed from memory
```

In this next example, we will expand our Grid class to hold point locations in two-dimensional arrays.

main.cpp

```
#include <iostream>
#include <string>
#include <vector>

using std :: cout;
using std :: endl;
using std :: string;
using std :: vector;

#define iLOOP for ( int i = 0 ; i < nx ; ++i )  // Macros for i/j looping
#define jLOOP for ( int j = 0 ; j < ny ; ++j )  //   in the 2-D grid.

class Grid
{
public:
  Grid(string _name, double x1, double x2, double y1, double y2, int _nx, int _ny)
  {
    name = _name;
    nx   = _nx;
    ny   = _ny;

    // Compute number of points in the x- and y-directions; grid spacing
    dx = (x2-x1)/(nx-1);
    dy = (y2-y1)/(ny-1);

    // Allocate memory
    px.resize(nx);
    py.resize(nx);

    iLOOP px[i].resize(ny);
    iLOOP py[i].resize(ny);

    //  Compute point locations
    for ( int i = 0 ; i < nx ; ++i )
      {
        for ( int j = 0 ; j < ny ; ++j )
          {
            px[i][j] = i*dx;
            py[i][j] = j*dy;
            }
      }
  }

  ~Grid() {  }

  string name;                 // Name of the grid
  int    nx;                   // No. pts. in the x-direction
  int    ny;                   // "    "   "   "  y     "
  double dx;                   // x-spacing
  double dy;                   // y-spacing
  vector<vector<double> > px;  // 2D array, storing x-coordinates of each mesh point
  vector<vector<double> > py;  // "   "       "     y      "      "   "     "

  void PrintInfo()
  {
    cout << "(o) Grid Information:                    " << endl;
    cout << "----------------------------------------" << endl;
    cout << "    (*) Name        : " << name            << endl;
    cout << "    (*) Num x points: " << nx              << endl;
    cout << "    (*) Num y points: " << ny              << endl;
  }
};

int main()
{
  Grid MyGrid("Square",0.,1.,0.,1.,10,10);
  MyGrid.PrintInfo();
}
```

The output of the code is shown below:

TTY Output

```
(o) Grid Information:
----------------------------------------
    (*) Name        : Square
    (*) Num x points: 10
    (*) Num y points: 10
```

The next example uses some techniques to make the code shorter and easier to read. We define a macro for the clumsy "vector of vectors" declaration for `px` and `py`. We also make use of the fact that semicolons, not linefeeds, are the delineators for lines of code. The above example, cleaned up using those tools, is shown below

main.cpp

```
#include <iostream>
#include <string>
#include <vector>

using std :: cout;
using std :: endl;
using std :: string;
using std :: vector;

#define iLOOP for ( int i = 0 ; i < nx ; ++i )  // Macros for i/j looping
#define jLOOP for ( int j = 0 ; j < ny ; ++j )  //   in the 2-D grid.
#define VD    vector<vector<double> >           // A 2D vector variable type
#define _D_   double                            // A double-precision scalar

class Grid
{
public:

  Grid(string _name, double x1, double x2, double y1, double y2, int _nx, int _ny)
  {
    name = _name;
    nx   = _nx;
    ny   = _ny;

    // Compute the grid spacing

    dx = (x2-x1)/(nx-1);    dy = (y2-y1)/(ny-1);

    // Allocate memory

             px.resize   (nx);    py.resize   (nx);
    iLOOP {  px[i].resize(ny);    py[i].resize(ny); }

    //  Compute point locations

    iLOOP jLOOP {  px[i][j] = i*dx;  py[i][j] = j*dy; }
  }

  ~Grid() {  }

  string name  ;   // Name of the grid
  int    nx, ny;   // No. pts. in the x- and y-directions
  _D_    dx, dy;   // x- and y-spacing
  VD     px, py;   // x- and y-coordinates of each mesh point

  void PrintInfo()
  {
    cout << "(o) Grid Information:                    " << endl;
    cout << "-----------------------------------------" << endl;
    cout << "    (*) Name        : " << name            << endl;
    cout << "    (*) Num x points: " << nx              << endl;
    cout << "    (*) Num y points: " << ny              << endl;
  }
};

int main()
{
  Grid MyGrid("Square",0.,1.,0.,1.,10,10);   MyGrid.PrintInfo();
}
```

Chapter 7

Python

7.1 Motivation

The motivation for this section is the same as for the C++ section. Specifically, we cover just enough material here to provide the "language" of this course. In other words, we spend some time introducing Python concepts and examples so that later, when we try to demonstrate how to implement key aspects of production code development, we have the tools to do so. As with the C++ section, this is by no means an exhaustive Python guidebook. But hopefully, it provides a helpful introduction not only for this class, but perhaps for technical work outside this class.

7.2 Basic Structure

Python is an interpreted language, i.e., not typically compiled. Rather, the Python software reads the program and executes it. Because it is interpreted, it is quicker to go from code modification to code execution. However, it is ultimately slower for big calculations when run in interpreted mode. Note that it is possible to compile Python, but we will not do that in this class.

Because it is an interpreted language, a Python program is often called a Python "script", and that may have also to do with the fact that the first line in a Python program can tell the operating system that Python is the software to be used to determinine it. For example, here is a small Python script:

example.py

```
#! /usr/bin/env python


def Example():

    print
    print "---------------------------------------------------------------------"
    print "|   PYTHON EXAMPLE                                                  |"
    print "---------------------------------------------------------------------"
    print

if __name__=='__main__':

    Example()        # Call the "Example" routine, above

```

And here is the command for running it:

Command line

```
./example.py
```

Notice the first line in the script. It tells the operating system that the contents of the file is to be interpreted by the Python software. So again, like with C-shell or Bash, we often call Python code Python “scripts.”

Proceding with this example, we encounter the word `def`. Here, we are defining a new function, or routine. In this case, we are defining the function `Example`. The `Example` function does not do much, except it prints to the screen a banner about the code. Notice that the print statements are indented. Indentation is a key aspect of Python. Whereas in C++, curly braces are used to begin and end program elements, including functions, if-statements, do-loops, etc., in Python it is the level of indentation that is used. So, the `Example` routine ends when the first level of indentation is disrupted.

And, that disruption happens in our example just below the last print statement, where we find an if-test that looks a bit misplaced. This if-test is present because it is possible that the capabilities in this file are being accessed not because we have entered the command `./main.py` but, rather, because they are being used by some other python package through the “import” process (discussed later). That import process is not occurring here. Rather, this script is being processed by Python by virture of the fact that someone has entered `./main.py` in the command line.

So, if-test at the bottom of the file evaluates to `True`, causing the next line to be executed, i.e., our `Example` routine is called. Like C++, it is important that the `Example` function precede any lines that call `Example` so that Python already knows about it.

Here is a summary of what happens in this first example:

1. We execute the script by typing `./main.py`

2. The operating system observes the first line, and uses the Python software to intepret our script.

3. The Python software notices we define a function called `Example` but does not execute it.

4. The Python software encounters our if-test and evaluates it as `True` because this file is being processed due to the fact the file is being executed directly (Item 1). Thus, `Example` is called.

5. `Example` does very little; it prints to the screen and ends, returning control to the (empty) lines after the if-test.

The output from this example is:

TTY Output

```
--------------------------------------------------------------------
|   PYTHON EXAMPLE                                                 |
--------------------------------------------------------------------
```

7.3 Lists

In this next example, we introduce the concept of Python lists. A list is like an array; it is a single variable that contains multiple values that can be accessed by index. But Python also allows for other ways to access entries besides using an index. Specifically, here is our code for this example:

example.py

```
#! /usr/bin/env python


def Example():

    print
    print "--------------------------------------------------------------------"
    print "|   PYTHON EXAMPLE                                                 |"
    print "--------------------------------------------------------------------"
    print

    A = [’apple’,’orange’,’cherry’]

    for a in A:
        print "Element of A = ",a

if __name__==’__main__’:

    Example()        # Call the "Example" routine, above
```

After the `A` list is defined, there is a `for` loop. The structure of the `for` loop is telling Python to consider, in turn, each entry in the `A` list. And, at the same time, as Python is considering each entry, the entry under consideration will be referred to by the variable `a` . Note that we could have chosen any letter to put there, i.e., `b` , or `AnEntryOfA` , etc. Then, inside the `for` loop, there is a print statement where the current entry under consideration is printed. Notice the indented print statement tells Python that it is part of the loop.

The command for running this example is identical to the previous example:

Command line

```
./example.py
```

Here is the output:

TTY Output

```
--------------------------------------------------------------------
|   PYTHON EXAMPLE                                                 |
--------------------------------------------------------------------

Element of A =  apple
Element of A =  orange
Element of A =  cherry
```

See how each element of the `A` list is printed in turn.

As with C++, it is possible to access elements of the `A` list using an index. This next example shows how:

example.py

```
#! /usr/bin/env python


def Example():

    print
    print "--------------------------------------------------------------------"
    print "|   PYTHON EXAMPLE                                                 |"
    print "--------------------------------------------------------------------"
    print

    A = ['apple','orange','cherry']

    print "This is how many elements are in A: ",len(A)

    for i in range(0,len(A)):
        print "A [" , i , "] = ",A[i]

    print "We can also print A all at once, like this: ", A

if __name__=='__main__':

    Example()        # Call the "Example" routine, above
```

We have introduced here the `len()` function, which returns the length of the argument. In this case, we use `len()` to return the length of our list. Next, the for-loop is still present in this example, but it uses a variable `i` that ranges from zero (the ID number of `A` 's first element to `2` , which is the ID of `A` 's last element. Lastly, we show in this example how to print all of the entries in `A` in one line. Here is the output:

TTY Output

```
--------------------------------------------------------------------
|   PYTHON EXAMPLE                                                 |
--------------------------------------------------------------------

This is how many elements are in A:  3
A [ 0 ] =  apple
A [ 1 ] =  orange
A [ 2 ] =  cherry
We can also print A all at once, like this:  ['apple', 'orange', 'cherry']
```

7.4 Dictionaries

A Python dictionary is similar to a Python list, except instead of using an index to refer to elements, a user-defined value is used. Consider the next example:

example.py

```
#! /usr/bin/env python


def Example():


    A = {’favorite fruit’:’apple’,’favorite veggie’:’aspargus’,’DOB’:’Jan. 12, 2000’}

    print "This is how many elements are in A: ",len(A)

    for a in A:
        print "A [" , a , "] = ",A[a]


if __name__==’__main__’:

    Example()        # Call the "Example" routine, above
```

Here,the setup of `A` is more complex. Notice that curly braces are used in the definition. Also, each entry must have two parts. The first part is the “index”, which instead of an integer is a value we choose. To be clear, the “indices” we choose are

- favorite fruit
- favorite veggie
- DOB

There is a special word for these user-defined indices, that word is “key”. So the above three are the “keys” in the dictionary. In the defintion of `A` , for each key there is an entry.

The `len()` function works just fine on the the dictionary; it handles lists, dictionaries, and many other data types. The for loop is similar to the previous example, except that we loop over the keys in `A` , and use each key as in “index” to extract the entry’s value. Here is the output:

TTY Output

```
This is how many elements are in A:  3
A [ favorite veggie ] =  aspargus
A [ DOB ] =  Jan. 12, 2000
A [ favorite fruit ] =  apple
```

7.5 Returning Values

In this next example, we modify our `Example` function to return values. At the same time, we show that it is possible to have a Python return multiple values, in this case a list *and* a dictionary. Here is the code:

example.py

```
#! /usr/bin/env python


def Example():

    A = [’apple’,’orange’,’cherry’]
    B = {’favorite fruit’:’apple’,’favorite veggie’:’aspargus’,’DOB’:’Jan. 12, 2000’}

    return A,B


if __name__==’__main__’:

    A_list, B_dict = Example()

    for i in range(0,len(A_list)):
        print "A_list [" , i , "] = ",A_list[i]

    for b in B_dict:
        print "B_dict [" , b , "] = ",B_dict[b]
```

and here is the output

TTY Output

```
A_list [ 0 ] =  apple
A_list [ 1 ] =  orange
A_list [ 2 ] =  cherry
B_dict [ favorite veggie ] =  aspargus
B_dict [ DOB ] =  Jan. 12, 2000
B_dict [ favorite fruit ] =  apple
```

Our `Example` routine returns `A` and `B` to the calling statement, which for us, is after the if-test. Notice how the call to `Example` has been modified so that it is prepared to receive, into variables `A_list` and `B_dict` , what `Example` provides. Notice how the names can be different in the caller side.

7.6 Handling Exceptions (Errors)

In this next example, we modify our `Example` function to return a single value by querying the `B` dictionary based on the request from the caling routine. Specifically, the calling routine passes a string argument to `Example` . In turn `Example` uses that argument to find the appropriate entry in `B` , and if found, returns that value. However, in `Example` we introduce a safegaurd for the situation in which the calling routine does not provide a legitimate entry. Here is the code:

example.py

```
#! /usr/bin/env python


def Example(DesiredEntry):

    B = {’favorite fruit’:’apple’,’favorite veggie’:’aspargus’,’DOB’:’Jan. 12, 2000’}

    try:
        return B[DesiredEntry]
    except:
        print "The entry you requested is not in the dictionary."
        exit(0)


if __name__==’__main__’:

    print ’Date of Birth:’
    val = Example(’DOB’)
    print val

    print ’Date of Birth: ’
    val = Example(’DoB’)
    print val
```

The `try-except` structure handles that situation by attempting to use the key provided by the user. If that key is found in the dictionary, all is well and `Example` returns the appropriate value. However, if anything goes wrong in the process, the `except` part of the error handler is executed.

In the calling routine, `Example` is called twice. The first time, a legitimate key is used (“DOB”). The second time, there is a typo, the “O” has been made lower case. That type causes the error handler to respond. Here is the output:

TTY Output

```
Date of Birth:
Jan. 12, 2000
Date of Birth:
The entry you requested is not in the dictionary.
```

7.7 Command-Line Options

In this next example, we make use of Python capabilities that enable the processing of command-line options. The code is shown here:

example.py

```
#! /usr/bin/env python

import sys
import getopt

def Example(argv):

    try:
         opts, args = getopt.getopt(argv,"h")   # Allowed options, for now -h (for help)
         print ’opts = ’,opts
         print ’args = ’,args
    except getopt.GetoptError:                  # Fatal error if user options are bad
        print "Error in user input options."
        exit(0)



if __name__==’__main__’:

    print ’*** Executing main routine. ***’

    args = sys.argv[1:]  # Retreives the command-line arguments that the user types in.
    Example(args)        # Passes command-line arguments to Example

    print ’--- Done. ---’
```

Notice the first few lines of the code, where we include the capabilities of the `sys` and `getopt` modules. These two modules provide functionality that we will use in this example.

Skipping down to the main routine, we have inserted print statements to display when the execution begins and when it completes. In between, we make use of the `sys.argv` functionality which returns a data structure containing the command-line arguments. That data structure is passed to our new `Example` function, above the main routine.

In `Example` an error handler is implemented to protect against an error that might arise when we use the `getopt` function to turn the command-line arguments into lists. If all goes well, `Example` prints those two lists. But if there is an error, it is because the command-line arguments do not satisfy the constraints that we place through the second argument to `getopt` , the “h” parameter. That second argument lists all of the acceptable command line options and arguments allowed by the script. The current example, only allows the “-h” (for help, supposedly) option. If the user tries using any other option, the error message is printed.

For this example, we execute the code three times:

Command line

```
./example.py
./example.py -h
./example.py -y
```

Here is the output:

TTY Output

```
*** Executing main routine. ***
opts =  []
args =  []
--- Done. ---
*** Executing main routine. ***
opts =  [(’-h’, ’’)]
args =  []
--- Done. ---
*** Executing main routine. ***
Error in user input options.
```

In the first execution, no command line arguments are given, which is fine. Note that the `opts` and `args` list is empty. In the second execution, the “-h” option is provided, and that option shows up in the `opts` list when printed. The third execution results in an error because the “-y” option is provided, and yet it is not an allowed option.

7.8 Command-Line Arguments

With very little change, the previous example demonstrates the use of arguments in addition to options. The code is shown here:

example.py

```
#! /usr/bin/env python

import sys
import getopt

def Example(argv):

    try:
        opts, args = getopt.getopt(argv,"h i:")
    except getopt.GetoptError:
        print "Error in user input options."
        exit(0)

    InputFilename = ’NONE’

    for opt, arg in opts:

        if opt == ’-h’:
            print ’(o) HELP: Besides -h for help, -i <input file name> is the only option.’
            sys.exit(0)
        elif opt in ("-i"):
            InputFilename = arg

    print "(o) The user has entered the following for the input file: " , InputFilename



if __name__==’__main__’:

    print
    print ’(o) Executing main routine.’

    args = sys.argv[1:]  # Retreives the command-line arguments that the user types in.
    Example(args)        # Passes command-line arguments to Example

    print ’(o) Done.’
```

The most important change is in the `getopt` line, where the code is passing the following argument to `getopt` :

`h i:`

The first `h` tells `getopt` that, as before, “-h” is an acceptable option. But it is now also including `i` so that “-i” is also a value option. However, the colon after `i` – note, the codes has `i:` tells `getopt` that this option also has an argument. This code’s loop over the options and arguments causes the if-test to be triggered for the `-i <input file>` option.

For this example, we run the code twice. Here is how we run it:

Command line

```
./example.py
./example.py -h
./example.py -y
```

Here is the output for those two runs:

TTY Output

```
*** Executing main routine. ***
opts =  []
args =  []
--- Done. ---
*** Executing main routine. ***
opts =  [(’-h’, ’’)]
args =  []
--- Done. ---
*** Executing main routine. ***
Error in user input options.
```

In the first execution, the request for help is made. In the second exection, the user supplies an input file. Note that the code takes the opportunity to store the input file in the variable `InputFilename` .

7.9 Opening and Reading a File

The previous example is extended slightly here to enable it to open the file specified by the user and read its contents. The code is shown on the next page.

example.py

```
#! /usr/bin/env python

import sys
import getopt

def Example(argv):

    try:
        opts, args = getopt.getopt(argv,"h i:")
    except getopt.GetoptError:
        print "Error in user input options."
        exit(0)

    InputFilename = ’NONE’

    for opt, arg in opts:

        if opt == ’-h’:
            print ’(o) HELP: Besides -h for help, -i <input file name> is the only option.’
            sys.exit(0)
        elif opt in ("-i"):
            InputFilename = arg

    print "(o) The user has entered the following for the input file: " , InputFilename

    print
    print
    print ’Echoing the input file...’
    print

    f = open(InputFilename,’r’)

    for line in f:
        print line.replace(’\n’,’’)

    f.close()



if __name__==’__main__’:

    print
    print ’(o) Executing main routine.’

    args = sys.argv[1:]  # Retreives the command-line arguments that the user types in.
    Example(args)        # Passes command-line arguments to Example

    print ’(o) Done.’
```

Note that the end-of-line characte must be removed (specifically, replaced with nothing) in order to avoid a double line-feed on printout. Here is how we run it:

Command line

```
./example.py -i MyInputFile
```

And here is the output, which the correct echo of the file that contains the lines printed:

TTY Output

```
(o) Executing main routine.
(o) The user has entered the following for the input file:  MyInputFile


Echoing the input file...


This is my input file.

It has information in it.

1
2
3
4
5
6
7
8


(o) Done.
```

7.10 Opening and Writing a File

Here we extent the previous example by writing the contents of the file specified by the user to another file, which is of the same name but with the string `.out` appended. Note the append operation in the file open command. The code is shown on the next page.

example.py

```
#! /usr/bin/env python

import sys
import getopt

def Example(argv):

    try:
        opts, args = getopt.getopt(argv,"h i:")
    except getopt.GetoptError:
        print "Error in user input options."
        exit(0)

    InputFilename = ’NONE’

    for opt, arg in opts:

        if opt == ’-h’:
            print ’(o) HELP: Besides -h for help, -i <input file name> is the only option.’
            sys.exit(0)
        elif opt in ("-i"):
            InputFilename = arg

    print "(o) The user has entered the following for the input file: " , InputFilename

    print
    print
    print ’Writing the input file contents to the output file’
    print

    f = open(InputFilename,’r’)
    g = open(InputFilename + ’.out’,’w’)

    for line in f:
        print line.replace(’\n’,’’)

        print >> g , line.replace(’\n’,’’)

    f.close()
    g.close()




if __name__==’__main__’:

    print
    print ’(o) Executing main routine.’

    args = sys.argv[1:]  # Retreives the command-line arguments that the user types in.
    Example(args)        # Passes command-line arguments to Example

    print ’(o) Done.’
```

Here is how we run it:

Command line

```
./example.py -i MyInputFile
```

Chapter 8

Nonlinearities, Stability, and Error Analysis

In this chapter we introduce basic methods for analyzing stability and anticipated order of error of a numerical scheme. We will stay with the finite difference method because it is the easiest to analyze.

8.1 Inuitive Explanation of Stability

Consider a 1D finite difference simulation of heat conduction. The governing equation is

$$\rho c_p \frac{\partial T}{\partial t} + \frac{\partial f}{\partial x} = 0 \tag{8.1}$$

There is a very effective and popular model that relates q to T known as Fourier's Law. It states simply that the heat flux is proportional to the gradient in temperature. Since heat flows from hot to cold, we need a negative sign. So, Fourier's Law is

$$f = -k \frac{\partial T}{\partial x} \tag{8.2}$$

If we substitute Fourier's Law into the governing equation, we get

$$\rho c_p \frac{\partial T}{\partial t} - k \frac{\partial^2 T}{\partial x^2} = 0 \tag{8.3}$$

which can then be discretized using the finite difference mehod as described in the Finite Difference chapter.

If we discretized it that way, and used the Forward Euler for time integration, the second term,

$$-k\frac{\partial^2 T}{\partial x^2} \tag{8.4}$$

would be evaluted at the previous time step, t^n. Or, put another way, f would be evaluated at t^n. Writing the fully discretized finite difference system in terms of q,

$$\rho c_p \frac{T_i^{n+1} - T_i^n}{\Delta t} + \frac{f_{i+1/2}^n - f_{i-1/2}^n}{\Delta x} = 0 \tag{8.5}$$

where

$$f_{i+1/2}^n = -k\frac{T_{i+1}^n - T_i^n}{\Delta t}$$
$$f_{i-1/2}^n = -k\frac{T_i^n - T_{i-1}^n}{\Delta t}$$

Solving for T^{n+1},

$$T_i^{n+1} = T_i^n + \frac{\Delta t}{\rho c_p}\frac{f_{i+1/2}^n - f_{i-1/2}^n}{\Delta x} \tag{8.6}$$

In the above three equations, the problem of stability are present. Consider, for example, a situation in which T_i^n, the center temperature, is less than its two neighbors. Physically, this would mean that heat would flow from the adjacent cells into cell i to help equalize the temperatures. Or, mathematically, it means that $f_{i+1/2}^n$ would be negative, meaning heat would from from $i+1$ to i, while $f_{i-1/2}^n$ would be positive, so heat would flow from $i-1$ to i.

This would all be correct, except there is one critical problem. If we use the situation at t^n , where we computed $f_{i+1/2}^n$ and $f_{i-1/2}^n$, *too long*, then we would not properly account for the fact that as heat flows into i, its temperature would increase, causing the q values to decrease. Put another way, we could use the situation at t^n to extrapolate too far forward in time and, as a result, T_i^{n+1} would exceed its neighbors at the end of the time step. Such a situation is not only unphysical, but it is unstable. Because that new center temperature, on the next time step, would give heat back to its neighbors, too much heat actually, because again the time step would be too large.

8.2 von Neumann Stability Analysis for Diffusion

Let us start with Equation 3.10, reduced to a one-dimensional problem with no heating term. Because it is one-dimensional only, the grid and the spatial derivative in the $y-$ direction is not present. There are no j subscripts as a result. Also, the coefficient 4 is reduced to 2. We are left with

$$C\frac{\phi_i^{n+1}-\phi_i^n}{\Delta t}-D\frac{\phi_{i+1}^n-2\phi_i^n+\phi_{i+1}^n}{h^2}=0 \tag{8.7}$$

or

$$\phi_i^{n+1}=\phi_i^n+\frac{C}{D}\frac{\Delta t}{h^2}\left(\phi_{i+1}^n-2\phi_i^n+\phi_{i+1}^n\right) \tag{8.8}$$

In heat conduction, the ratio of C/D is called thermal diffusivity and is often represented by α. We will do that here,

$$\phi_i^{n+1}=\phi_i^n+\alpha\frac{\Delta t}{h^2}\left(\phi_{i+1}^n-2\phi_i^n+\phi_{i+1}^n\right) \tag{8.9}$$

The approach to stability analysis is to determine what the above equation will do to an initial distribution in $\phi(x)$. Without a forcing function, q, any disturbance in $\phi(x)$ should dissipate over time. If it grows, then not only is the method inaccurate, it is unstable. So, we mathematically introduce an initial condition in $\phi(x)$ and then try to determine analytically what the algorithm as embodied in Equation 8.9 will do to that initial condition.

We will introduce sinusoidal functions into the solution field as an initial condition, i.e., for the solution field ϕ at t^n. Recall Euler's formula,

$$e^{ix}=\cos x+i\sin x \tag{8.10}$$

Thus, sinusoids can be represented with e^{ik_mx}, where k_m is the frequency. Using e^{ik_mx} in Equation 8.9 is a convenient way to introduce the sinusoidal functions using a single, easily manipulated term. Furthermore, because Equation 8.9 is linear, we can consider each frequency k_m separately and add the results. Or, to be more clear, we only need to consider the generic e^{ik_mx}.

Since we are only considering the generic e^{ik_mx}, we are seeking what we will call $\phi_m(x,t)$ as the algorithm's result of that initial signal. We need a way to track the growth of the initial signal, and we will do that by proposing an amplication factor multiplied by the initial signal. In other words, we propose that

$$\phi_m(x,t)=a_m(t)e^{ik_mt} \tag{8.11}$$

and we wish to know how $a_m(t)$ behaves over time. If it decreases, then the initial signal is decaying. If it grows, then the initial signal is growing and the algorithm is unstable. For the amplification term $a_m(t)$, we also need to propose some form. We need the form to satisfy the initial condition

$$\phi_m(x,0) = a_m(0)e^{ik_m x} = e^{ik_m x} \tag{8.12}$$

i.e., we need

$$a_m(0) = 1 \tag{8.13}$$

There are many functions that would satisfy that condition. In von Neumann stability analysis, we use the form

$$a_m(t) = e^{ct} \tag{8.14}$$

So, we are therefore seeking the form

$$\phi_m(x,t) = e^{ct} e^{ik_m x} \tag{8.15}$$

Note that c may be complex.

To determine if this form is useful, let us substitute it into our algorithm, Equation 8.9. Before doing that, we constrain our analysis to use a constant time step Δt, so that time at timestep n is $t^n = n\Delta t$. We get

$$e^{c(t+\Delta t)} e^{ik_m x} - e^{ct} e^{ik_m x} = \alpha \frac{\Delta t}{h^2} \left(e^{ct} e^{ik_m (x-h)} - 2e^{ct} e^{ik_m x} + e^{ct} e^{ik_m (x+h)} \right) \tag{8.16}$$

Dividing the above equation by $e^{ct} e^{ik_m x}$ results in

$$e^{c\Delta t} - 1 = \alpha \frac{\Delta t}{h^2} \left(e^{-ik_m h} - 2 + e^{ik_m x} \right) \tag{8.17}$$

Next we make use of the mathematical relation

$$\cos\theta = \frac{e^{i\theta} + e^{-i\theta}}{2} \tag{8.18}$$

to combine to of the terms on the right hand side. We get

$$e^{c\Delta t} - 1 = 2\alpha \frac{\Delta t}{h^2} \left[\cos(k_m h) - 1 \right] \tag{8.19}$$

or

$$e^{c\Delta t} = 1 + 2\alpha \frac{\Delta t}{h^2} \left[\cos(k_m h) - 1 \right] \tag{8.20}$$

Next, we make use of the trigonometric identity,

$$\sin^2\frac{\theta}{2} = \frac{1-\cos\theta}{2} \tag{8.21}$$

on the right-hand side, to get

$$e^{c\Delta t} = 1 - 4\alpha\frac{\Delta t}{h^2}\left[\sin^2\left(\frac{k_m h}{2}\right)\right] \tag{8.22}$$

The above equation is the amplification term we originally sought, i.e.,

$$a_m(\Delta t) = 1 - 4\alpha\frac{\Delta t}{h^2}\left[\sin^2\left(\frac{k_m h}{2}\right)\right] \tag{8.23}$$

For the algorithm to be stable, we need the magnitude of $a_m(t)$ to never be greater than 1, i.e., we need

$$\left|1 - 4\alpha\frac{\Delta t}{h^2}\left[\sin^2\left(\frac{k_m h}{2}\right)\right]\right| < 1 \tag{8.24}$$

When considering this inequality, first consider the possibility that the term inside absolute values is positive. Then the absolute values signs are not needed, and we would have the requirement

$$1 - 4\alpha\frac{\Delta t}{h^2}\left[\sin^2\left(\frac{k_m h}{2}\right)\right] < 1 \tag{8.25}$$

Because the 1s on each side cancel, this condition would always be satisfied. Now, consider the possibility that the term inside the absolute values is negative, We would have the requirement

$$4\alpha\frac{\Delta t}{h^2}\left[\sin^2\left(\frac{k_m h}{2}\right)\right] - 1 < 1 \tag{8.26}$$

Here the 1s do not cancel, and instead we have

$$4\alpha\frac{\Delta t}{h^2}\left[\sin^2\left(\frac{k_m h}{2}\right)\right] < 2 \tag{8.27}$$

The sinusoid squared is never greater than unity, thus simply replacing it with 1 means we obtain the requirement

$$\alpha\frac{\Delta t}{h^2} < \frac{1}{2} \tag{8.28}$$

or

$$\Delta t < \frac{h^2}{2\alpha} \tag{8.29}$$

Notice that the original frequency term k_m is not part of this result. This equation states limitations on the time step for the explicit finite difference formulation applied to the diffusion equation. For materials that are more diffuse, with less resistance to diffusivity, α will be larger. A larger α will require smaller timesteps. Also, a finer mesh requires smaller timesteps.

8.3 The Courant Number for the Wave Equation

We can pursue the same approach to stability analysis for other discretization methods [1] [4]. One of the most important in a multi-physics code is for the motion of material in explicit time marching methods. For that purpose, we consider the momentum component of the Navier-Stokes equatioins. In one-dimension, and with no viscosity term, we have a wave equation of the form

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{8.30}$$

We will consider the fully discrete method proposed by Lax, where the velocity at the old time step is the average of cell's old values on either side, i.e., we set

$$u_j^n = \frac{u_{j+1}^n + u_{j-1}^n}{2} \tag{8.31}$$

Using that in a Forward Euler time marching scheme, with a central difference approximation for the spatial derivative, the PDE is discretized as

$$\frac{u_j^{n+1} - \frac{u_{j+1}^n + u_{j-1}^n}{2}}{\Delta t} - c\left(\frac{u_{j+1}^n - u_{j-1}^n}{2h}\right) = 0 \tag{8.32}$$

Solving for u_j^{n+1},

$$u_j^{n+1} = \frac{u_{j+1}^n + u_{j-1}^n}{2} - \Delta t c\left(\frac{u_{j+1}^n - u_{j-1}^n}{2h}\right) \tag{8.33}$$

As before, we introduce a sinusoidal initial condition along with an amplification factor to quantify how the signal would evolve, i.e., we seek

$$u(x,t) = e^{bt}e^{ik_m x} \tag{8.34}$$

Substituting that into Equation 8.33 for $u = u(x,t)$,

$$e^{b(t+\Delta t)}e^{ik_mx} = \frac{e^{bt}e^{ik_m(x+h)} + e^{bt}e^{ik_m(x-h)}}{2} - \Delta t c\left(\frac{e^{bt}e^{ik_m(x+h)} - e^{bt}e^{ik_m(x-h)}}{2h}\right) \tag{8.35}$$

Divide out $c^{bt}e^{ik_mx}$,

$$e^{b\Delta t} = \frac{e^{ik_mh} + e^{-ik_mh}}{2} - \frac{\Delta t c}{h}\left(\frac{e^{ik_mh} - e^{-ik_mh}}{2}\right) \tag{8.36}$$

Making use again of the identity in Equation 8.18 along with the companion identity

$$\sin\theta = \frac{e^{i\theta} - e^{-i\theta}}{2i} \tag{8.37}$$

we obtain

$$e^{b\Delta t} = \cos(k_mh) - \frac{\Delta t c}{h} i\sin(k_mh) \tag{8.38}$$

which, as before, is our “amplification” term. For the algorithm to be stable, we need it to be less than unity. In other words, we must require that

$$\left|\cos(k_mh) - \frac{\Delta t c}{h} i\sin(k_mh)\right| \leq 1 \tag{8.39}$$

or, writing out the magnitude of the complex number on the left,

$$\sqrt{\cos^2(k_mh) + \left(\frac{\Delta t c}{h}\right)^2 \sin^2(k_mh)} \leq 1 \tag{8.40}$$

If the additional factor inside the radical were not present, we would have

$$\sqrt{\cos^2(k_mh) + \sin^2(k_mh)} = 1 \tag{8.41}$$

The additional multiplicative factor serves to increase the magnitude if it is greater than unity. Thus, we must require that

$$\frac{\Delta t c}{h} \leq 1 \tag{8.42}$$

The value on the left is called the Courant number,

$$\mathrm{Co} \equiv \frac{\Delta t c}{h} \tag{8.43}$$

which, again, must be less than unity for the scheme to be stable. Note that this is a necessary condition, not necessarily a sufficient condition.

Other methods may be used to compute the Navier-Stokes equations, and an stability analysis may or may not have been carried out for them. Still, the *idea* of the Courant condition is often considered to apply directly when the scheme is explicit in time. Multiplicative factors may be used as factors of safety and, also, to accommodate peculiarities in the scheme being used.

With that in mind, it is helpful to contemplate what the Courant number repesents. If c is the wavespeed, then Δtc is the distance traveled by a disturbance during one timestep. Notice that distance is divided by the length of the cell, and we want that quotient to be less than unity. In other words, the condition is stating that we do not want information to be able to skip over a cell in a single timestep. That intuitive interpretation enables the concept of the Courant condition to be applied in multiple dimensions without additional analysis. With some trial and error, it can be used effectively for other schemes, with mutliple cell sizes, etc. Typically, there is a single cell in an actual application where its size and the wavespeed, i.e., speed of sound, of the material in it restricts the size of the time step. Typically, time step sizes change dynamically during applications. At each time step, the Courant conditioin is computed on each cell and the minimum across the entire mesh is taken, again, sometiems with a factor of safety.

8.4 Error Estimation and the Taylor Series

Taylor series expansion provide the foundation for the error estimation methods discussed here. The Taylor series with the remainder term is

$$\begin{aligned} f(x) = f(x_o) &+ \left.\frac{df}{dx}\right|_{x_o} (x - x_o) \\ &+ \frac{1}{2!} \left.\frac{d^2 f}{dx^2}\right|_{x_o} (x - x_o)^2 \\ &+ \frac{1}{3!} \left.\frac{d^3 f}{dx^3}\right|_{x_o} (x - x_o)^3 \\ &+ \ldots \\ &+ \frac{1}{k!} \left.\frac{d^k f}{dx^k}\right|_{x_o} (x - x_o)^k \\ &+ \frac{1}{(k+1)!} \left.\frac{d^{k+1} f}{dx^{k+1}}\right|_{\xi} (\xi - x_o)^k \end{aligned} \tag{8.44}$$

where ξ is between x and x_o. Notice that this is an exact equality because of the presence of the last term, where the exact position of ξ is unspecified.

As an aside, the derivation of the infinite form of the Taylor series, works as follows. We seek coefficients a_o, a_i, a_2, etc., such that

$$f(x) = a_o + a_1(x - x_0) + a_2(x - x_0)^2 + a_3(x - x_0)^3 + \ldots \tag{8.45}$$

We can determine a_o quite easily simply by setting $x = x_o$ in the above expression. We get

$$f(x_o) = a_o \tag{8.46}$$

Finding a_1 is almost as easy. We take the first derivative of Equation 8.45 and get

$$\frac{df(x)}{dx} = 0 + a_1 + 2a_2(x - x_0) + 3a_3(x - x_0)^2 \ldots \tag{8.47}$$

Again, we simply set $x = x_0$ in this equation to get

$$\left.\frac{df(x)}{dx}\right|_{x = x_o} = a_1 \tag{8.48}$$

Finding a_2 is accomplished in the same way, except by taking the second derivative of Equation 8.45,

$$\frac{d^2f(x)}{dx^2} = 0+0+2a_2+3\cdot 2a_3(x-x_0)+... \tag{8.49}$$

and again setting $x = x_0$, or

$$\left.\frac{d^2f(x)}{dx^2}\right|_{x=x_0} = 0+0+2a_2 \tag{8.50}$$

The procedure can be repeated indefinitely. Notice in Equation 8.49 the $3\cdot 2$ coefficient. Even though that term is zeroed out in the above step, the $3\cdot 2$ is a foreshadowing of what will occur when solving for a_3, where the *third* derivative will be required. The coefficient on a_3 will, in fact be $3\cdot 2$, which is the same as 3 factorial (3!).

8.5 Expressing Finite Differences using Taylor Series

Consider the one-dimensional steady state convection-diffusion equation

$$-\frac{d^2\phi}{dx^2}+u\frac{d\phi}{dx} = 0 \tag{8.51}$$

Previously, in the chapter on finite differences, we had simply introduced approximations of the derivatives. We wish to develop an *exact* representation of the above classical statement on a finite difference grid, with spacing h, i.e., with points $x_i, i = 1,2,3...N$ each separated by a distance h.

We will use the remainder form of the Taylor series expansion for each of the two terms in our differential equation, and we will expand about each grid point, x_i. First, we write an exact representation of $\phi(x)$ between grid point x_i and its neighbor to the right:

$$\phi(x_i+h) = \phi(x_i)+h\left.\frac{d\phi}{dx}\right|_{x_i}+\frac{h^2}{2!}\left.\frac{d^2\phi}{dx^2}\right|_{x_i}+\frac{h^3}{3!}\left.\frac{d^3\phi}{dx^3}\right|_{x_i}+\frac{h^4}{4!}\left.\frac{d^4\phi}{dx^4}\right|_{\xi_r} \tag{8.52}$$

where ξ_r is between the grid point x_i and the grid point on the right, x_{i+1}. Second, we write the analogous representation of $\phi(x)$ between x_i and its left neighbor:

$$\phi(x_i-h) = \phi(x_i)-h\left.\frac{d\phi}{dx}\right|_{x_i}+\frac{h^2}{2!}\left.\frac{d^2\phi}{dx^2}\right|_{x_i}-\frac{h^3}{3!}\left.\frac{d^3\phi}{dx^3}\right|_{x_i}+\frac{h^4}{4!}\left.\frac{d^4\phi}{dx^4}\right|_{\xi_l} \tag{8.53}$$

where ξ_l is between x_i and the grid point on the left. Adding these two equations together, the odd derivatives cancel and we have

$$\phi(x_i - h) + \phi(x_i + h) = 2\phi(x_i) + h^2 \left.\frac{d^2\phi}{dx^2}\right|_{x_i} + \frac{h^4}{4!}\left(\left.\frac{d^4\phi}{dx^4}\right|_{\xi_l} + \left.\frac{d^4\phi}{dx^4}\right|_{\xi_r} \right) \tag{8.54}$$

From this equation, we can solve for the second derivative directly and exactly to get

$$\left.\frac{d^2\phi}{dx^2}\right|_{x_i} = \frac{\phi(x_i - h) - 2\phi(x_i) + \phi(x_i + h)}{h^2} - \frac{h^2}{4!}\left(\left.\frac{d^4\phi}{dx^4}\right|_{\xi_l} + \left.\frac{d^4\phi}{dx^4}\right|_{\xi_r} \right) \tag{8.55}$$

Assuming $\phi(x)$ is sufficiently continuous, application of the Intermediate Value Theorem yields

$$\left.\frac{d^2\phi}{dx^2}\right|_{x_i} = \frac{\phi(x_i - h) - 2\phi(x_i) + \phi(x_i + h)}{h^2} - \frac{h^2}{3!}\left.\frac{d^4\phi}{dx^4}\right|_{\xi} \tag{8.56}$$

In Equation 8.56, we have an exact representation of the solution's second derivative consisting of a finite difference form and an error term. Granted, the error term is not known, but it is useful for error analysis, as will be seen shortly.

First, though, we must form an exact representation of the first derivative as well. Analogous to what we did for the second derivative, we have

$$\phi(x_i + h) = \phi(x_i) + h\left.\frac{d\phi}{dx}\right|_{x_i} + \frac{h^2}{2!}\left.\frac{d^2\phi}{dx^2}\right|_{x_i} + \frac{h^3}{3!}\left.\frac{d^3\phi}{dx^3}\right|_{\eta_r} \tag{8.57}$$

and

$$\phi(x_i - h) = \phi(x_i) - h\left.\frac{d\phi}{dx}\right|_{x_i} + \frac{h^2}{2!}\left.\frac{d^2\phi}{dx^2}\right|_{x_i} - \frac{h^3}{3!}\left.\frac{d^3\phi}{dx^3}\right|_{\eta_l} \tag{8.58}$$

where η_l and η_r are analogous to their ξ counterparts, above. Subtracting these two equations,

$$\phi(x_i + h) - \phi(x_i - h) = 2h\left.\frac{d\phi}{dx}\right|_{x_i} + \frac{h^3}{3!}\left(\left.\frac{d^3\phi}{dx^3}\right|_{\eta_l} + \left.\frac{d^3\phi}{dx^3}\right|_{\eta_r} \right) \tag{8.59}$$

and then solving for the first derivative, we get

$$\left.\frac{d\phi}{dx}\right|_{x_i} = \frac{\phi(x_i + h) - \phi(x_i - h)}{2h} - \frac{h^2}{2 \cdot 3!}\left(\left.\frac{d^3\phi}{dx^3}\right|_{\eta_l} + \left.\frac{d^3\phi}{dx^3}\right|_{\eta_r} \right) \tag{8.60}$$

Again, by appealing to the Intermediate Value Theorem,

$$\left.\frac{d\phi}{dx}\right|_{x_i} = \frac{\phi(x_i+h)-\phi(x_i-h)}{2h} - \frac{h^2}{3}\left(\left.\frac{d^3\phi}{dx^3}\right|_{\eta}\right) \tag{8.61}$$

Combining Equations 8.61 and 8.56, we can write the original differential equation in exact finite difference form as follows

$$-\left(\frac{\phi(x_i-h)-2\phi(x_i)+\phi(x_i+h)}{2}\right)+\frac{\phi(x_i+h)-\phi(x_i-h)}{2h} = \frac{h^2}{3}\left(\left.\frac{d^3\phi}{dx^3}\right|_{\eta}\right)-\frac{h^2}{12}\left.\frac{d^4\phi}{dx^4}\right|_{\xi} \tag{8.62}$$

or

$$-\left(\frac{\phi(x_i-h)-2\phi(x_i)+\phi(x_i+h)}{2}\right)+\frac{\phi(x_i+h)-\phi(x_i-h)}{2h} = h^2\left(\frac{1}{3}\left.\frac{d^3\phi}{dx^3}\right|_{\eta}-\frac{1}{12}\left.\frac{d^4\phi}{dx^4}\right|_{\xi}\right) \tag{8.63}$$

Writing this as a finite difference method, i.e., using subscripts for grid point values,

$$-\left(\frac{\phi_{i-1}-2\phi_i+\phi_{i+1}}{2}\right)+u\frac{\phi_{i+1}-\phi_{i-1}}{2h} = h^2\left(\frac{1}{3}\left.\frac{d^3\phi}{dx^3}\right|_{\eta}-\frac{1}{12}\left.\frac{d^4\phi}{dx^4}\right|_{\xi}\right) \tag{8.64}$$

On the left of the equation above, we have the finite difference formulation of the convection-diffusion problem. On the right, we have an unknown error term. Notice the exact equality. While we do not know the parenthetical term exactly, we can say that the leading coefficient, h^2 will decrease as h is refined. Thus we say that the numerical method is spatially *convergent*. Moreover, we say that the error is *second order*. The finite difference method as derived here is also therefore referred to as being a second order method. It is worth noting that the exponent in h is a key value. By make use of additional grid points' values, e.g., using x_{i-2} and x_{i+2}, we could eliminate the error terms above in favor of higher order terms. In essence, we would be constructing a higher-order method.

8.6 Sub-Cycling

In a multi-physics simulation, or even a simpler simulation that involves complex constitutive models, there may be multiple time scales involved. For example, consider the Prandtl-Reuss equation for plasticity, Equation . It is a constitutive model that must be integrated in time along with the equations of motion and the energy evolution equation. However, the time step needed to ensure a desired level of accuracy in that equation is not necessarily of the same order as the time step otherwise required by the PDEs, e.g., the “host code”. There are at least two approaches for dealing with this eventuality in a multi-physics code:

1. Develop a method inside the code that queries every component, every equation solver, and every model for their maximum recommended timestep; note that the timestep might change as the simulation progresses. Therefore, the query must be made often, if not at every time step. Choose the minimum time step among all of the responses and proceed with the calculation that way.

2. Combine the first approach with sub-cycling. If there are multiple orders of magnitude in the responses from various physics packages and models, an algorithm can be designed to determine which time step to use and then sub-cycle algorithms that have extremely small time step requirements.

8.7 Nonlinear Problems and Solvers

There are numerous examples of nonlinearities that affect the solution methods in computational physics. Already in the discussion so far of the continuum equations for mechanics there are nonlinearities, specifically the convective term for energy transport and momentum transport present nonlinearities. But oftentimes constitutive models are dependent on the state variables, and boundary conditions can be nonlinearly dependent. There are multiple ways of dealing with nonlinearites, far too many to discuss here. It is a broad and complex subject, but here we discuss a few.

8.7.1 Explicit Time Marching

Perhaps the simplest approach to dealing with a nonlinearity is to use an explicit time marching method, where the nonlinearities are evaluated at the previous time step. For example, if in the heat conduction equation the thermal conductivity and specific heat were functions of temperature, then the governing equation would be

$$\rho c_p(T)\frac{\partial T}{\partial t}+\frac{\partial}{\partial x}\left(k(T)\frac{\partial T}{\partial x}\right)=0 \qquad (8.65)$$

If the finite difference or finite method were applied to this system, it would result in a nonlinear set of simultaneous equations,

$$\mathbf{C}\frac{\partial}{\partial t}\mathbf{T}+\mathbf{K}(\mathbf{T})\mathbf{T}=\mathbf{0} \tag{8.66}$$

where $\mathbf{T}$ is the vector of unknown temperature values and $\mathbf{C}$ and $\mathbf{K}$ are matrices with coefficients that depend on $\mathbf{T}$. Note that the finite difference method produces a $\mathbf{C}$ that is diagonal but the finite element method does not. It produces a sparse but non-diagonal $\mathbf{C}$.

In an explicit, Forward Euler, method, we would write

$$\mathbf{C}\mathbf{T}^{n+1}=\mathbf{C}\mathbf{T}^{n}+\Delta t\mathbf{K}(\mathbf{T}^{n})\mathbf{T}^{n} \tag{8.67}$$

and the nonlinearity is removed. Despite its very simple approach, the explicit method of integrating systems in this way is very popular because there is no matrix inversion required and no nonlinear solve required.

8.7.2 Nonlinear Lagging

If time step restrictions are too great with the explicit approach above, an implicit time marching method may be advantageous. If the "theta method", or Crank-Nicholson method (when $\theta=1/2$) is used, the system would be written as

$$\left(\mathbf{C}(\mathbf{T})+\Delta t(1-\theta)\mathbf{A}(\mathbf{T})\right)\mathbf{T}^{n+1}=-\theta\Delta t\mathbf{A}(\mathbf{T})\mathbf{T}^{n}+\mathbf{C}(\mathbf{T})\mathbf{T}^{n}+\mathbf{q} \tag{8.68}$$

or more compactly as

$$\mathbf{K}(\mathbf{T})\mathbf{T}^{n+1}=\mathbf{f}(\mathbf{T})+\mathbf{g}\mathbf{T}^{n} \tag{8.69}$$

Here we see the nonlinearity appear in the matrices. If we were to evaluate those matrices using the temperature values from the previous timestep, the system would be "linearized", specifically, we would solve

$$\mathbf{K}(\mathbf{T}^{n})\mathbf{T}^{n+1}=\mathbf{f}(\mathbf{T}^{n})+\mathbf{g}\mathbf{T}^{n} \tag{8.70}$$

The stability analysis in the previous sections might not stricly apply, depending on the severity of the nonlinearity.

8.7.3 Successive Approximation

When it is desirable to take the larger time steps afforded by an implicit method, but the problem is nonlinear and nonlinear lagging produces unacceptable results, the nonlinear system (Equation 8.69) can be solved by iteration. To notate the iteration number, here we will introduce a subscript i, i.e., $\mathbf{T}_i^{n+1}$ is the ith iterate in a nonlinear iteration scheme focused on finding the solution $\mathbf{T}^{n+1}$ at the $n+1$ timestep.

In the successive approximation approach, we can use the previous timestep's solution as the initial guess, i.e., set

$$\mathbf{T}_0^{n+1} = \mathbf{T}^{n+1} \tag{8.71}$$

Then we evaluate the nonlinear terms using that initial guess to form the new guess, repeating the process until convergence is reached. To be clear, then, at each Successive Approximation iteration, we solve

$$\mathbf{K}(\mathbf{T}_i^{n+1})\mathbf{T}_{i+1}^{n+1} = \mathbf{f}(\mathbf{T}_i^n) + \mathbf{g}\mathbf{T}^n \tag{8.72}$$

for $i = 1,2,3...$ until convergence is reached. In mechanics applications, the Successive Approximation method typically allows for a wide range of initial guesses that still lead to convergence, i.e., it has a large "domain of convergence."

8.7.4 Newton-Raphson Iteration

The Newton-Raphson method is an extension of Newton's method that was developed for finding the root of a nonlinear function. For a nonlinear function $f(x)$, Newton's method seeks x such that $f(x) = 0$. Given the current iterate, x_i, f and f' are evaluated at that iterate. Those two values are used to form the line that is tangent to $f(x)$ at x_i. The root of that line is the easily found and made the next iterate, x_{i+1}. The equation that must be solved for x_{i+1} is

$$f'(x_i)\delta_{i+1} = -f(x_i) \tag{8.73}$$

which is straightforward, then $x_{i+1} = x_1 + \delta_{i+1}$

When the nonlinear function is a vector function, and a function of another vector, the Newton-Raphson iteration is a direct extension of the one above, but instead of the scalar derivative, it uses the extention of that, which is the Jacobian, $\mathbf{J}$ whose entries are

$$J_{ij} = \frac{\partial f^i}{\partial x^j} \tag{8.74}$$

where x^j is the jth component of $\mathbf{x}$ and f^i is the ith component of $\mathbf{f}$. The Newton-Raphson iteration is

$$\mathbf{J}(\mathbf{x}_i)\boldsymbol{\delta}_{i+1} = -\mathbf{f}(\mathbf{x}_i) \tag{8.75}$$

which must be inverted by a linear system solver, and then $\mathbf{x}_{i+1} = \mathbf{x}_i + \delta_i$

Chapter 9

Method of Manufactured Solutions

The method of manufactured solutions is a straightforward, but powerful and flexible way to verify computational physics codes are solving their governing equations correctly. A good way to explain the method is to consider a model problem, such as

$$\frac{\partial \phi}{\partial t} - D\nabla^2 \phi = 0 \tag{9.1}$$

Suppose we have developed a discrete method for this PDE, using finite elements or finite differences, for example, which result in a semi-discrete system

$$\mathbf{C}\frac{d\boldsymbol{\phi}}{dt} - \mathbf{D}\boldsymbol{\phi} = \mathbf{0} \tag{9.2}$$

or perhaps a fully discretized version using Forward Euler

$$\mathbf{C}\boldsymbol{\phi}^{n+1} = \mathbf{C}\boldsymbol{\phi}^{n} + \Delta t \mathbf{D}\boldsymbol{\phi}^{n} \tag{9.3}$$

We want to verify that **C** and **D** are formed correctly and that we have implemented the time-marching scheme correctly. Also, we would like to verify that the boundary conditions for this system are implemented correctly. Suppose that our implementation can accommodate two types of boundary conditions:

1. Fixed BCs: $\phi(x,y,t) = \phi_o x,y,t$, a known function.

2. Flux BCs: $\nabla\phi \cdot \mathbf{n} = q(x,y,t)$, a known function.

The discrete method we have build can apply only the first type, only the second type, or both types on different parts of the boundary.

In the method of manufactured solutions we modify the left-hand side of the PDE, Equation 9.1, such that the solution to the PDE is a function of our choossing.

9.1 Example 1

Suppose we want the solution to be

$$\phi = t \cdot (x+y) \tag{9.4}$$

If we add the appropriate right-hand side function, $f(x,y,t)$ to Equation 9.1 (and apply the correct boundary conditions) we can indeed cause Equation 9.4 to be the solution. So, we modify the model problem to read

$$\frac{\partial \phi}{\partial t} - D\nabla^2 \phi = f(x,y,t) \tag{9.5}$$

To find $f(x,y,t)$, we substitute our desired solution from Equation 9.4 into Eqauation 9.5:

$$\frac{\partial}{\partial t}\left(t \cdot (x+y)\right) - D\nabla^2 \left(t \cdot (x+y)\right) = f(x,y,t) \tag{9.6}$$

Carrying out the partial derivates on the left-hand side,

$$(x+y) - D\,(0+0) = f(x,y,t) \tag{9.7}$$

or

$$f(x,y,t) = (x+y) \tag{9.8}$$

So, this problem:

$$\frac{\partial \phi}{\partial t} - D\nabla^2 \phi = (x+y) \tag{9.9}$$

with fixed BCS around the entire domain of $\phi = t(x+y)$ has the solution in Equation 9.4.

9.2 Example 2

The previous example tested our **D** and **C** matrices and tested the coding that applies fixed boundary conditions. Now suppose we also want to test our flux boundary condition coding. If this problem were on the unit square, i.e., $0 \geq x,y \leq 1$, then we could state a second test problem as follows:

$$\frac{\partial \phi}{\partial t} - D\nabla^2 \phi = (x+y) \tag{9.10}$$

with the following BCs:

- $\phi = t \cdot (x+y)$ on boundaries $x = 0, y = 0, y = 1$, and
- $\frac{\partial \phi}{\partial x} = t$ on boundary $x = 1$

Note that second BC. The normal on that surface is **i** and $\nabla\phi = \nabla(t \cdot (x+y)) = t\mathbf{i} + t\mathbf{j}$, so $\nabla\phi \cdot \mathbf{n} = t$. By using both types of boundary conditions more coding can be tested and the code should yield exactly the same answer.

9.3 Example 3

To sure up ideas, we consider a slightly more complex desired solution,

$$\phi(x,y,t) = e^{-t}\left(3x^2y + 5y^2x\right) \tag{9.11}$$

We first endeavor to find our function $f(x,y,t)$ that would cause

$$\frac{\partial \phi}{\partial t} - D\nabla^2\phi = f(x,y,t) \tag{9.12}$$

to have that solution. So, we insert it into the left-hand side,

$$\begin{aligned} &\frac{\partial}{\partial t}\left[e^{-t}\left(3x^2y + 5y^2x\right)\right] \\ -D&\left[\frac{\partial^2}{\partial x^2}\left(e^{-t}\left(3x^2y + 5y^2x\right)\right) + \frac{\partial^2}{\partial y^2}\left(e^{-t}\left(3x^2y + 5y^2x\right)\right)\right] \\ &= f(x,y,t) \end{aligned} \tag{9.13}$$

or

$$-e^{-t}\left(3x^2y + 5y^2x + 16y\right) = f(x,y,t) \tag{9.14}$$

So, our problem statement for this test case is

$$\frac{\partial \phi}{\partial t} - D\nabla^2\phi = -e^{-t}\left(3x^2y + 5y^2x + 16y\right) \tag{9.15}$$

with whatever choices of boundary conditions we want, for example,

- $\phi = e^{-t}\left(3x^2y + 5y^2x\right)$ on some boundaries and
- $\nabla \cdot \left(e^{-t}\left(3x^2y + 5y^2x\right)\right) \cdot \mathbf{n}$ on others, at our discretion

9.4 Nonlinear Problems

The method of manufactured solutions can be applied very effectively to root out bugs and verify nonlinear solvers. Consider this model problem

$$\frac{\partial \phi}{\partial t} - \nabla \cdot (D(\phi)\nabla \phi) = 0 \tag{9.16}$$

As usual, we choose our desired solution, which here we will call $\phi_a(x,y,t)$ and we seek a right-hand side $f(x,y,t)$ that will cause the new PDE to have that solution.

In many applications, $D(\phi)$ is a look-up table, not an analytic function. Or, in some applications, it might be the result on an algorithm. In the tabular look-up case, a table could be created that has a known functional relationship for $D(\phi)$, say $D = D_a(\phi)$. Then, using our ϕ_a, a table of $D_a(\phi_a)$ values could be created. At the same time, that $D_a(\phi_a)$ could be substituted into the model problem for $D(\phi)$ and the $f(x,y,t)$ function computed.

Once an $f(x,y,t)$ function is obtained that is consistent with the treatment of the nonlinear term, the code can be run with that f, along with the boundary conditions of choice. Then, when the code is running, there are two options for evaluating the nonlinear term:

1. Provide the correct answer so that $D_a = D_a(\phi)a$ always provides the correct coefficient. In so doing, the nonlinear is effectively removed, and replaced with a time- and space-varying coefficient.n

2. Allow ϕ to "float" with the solver, i.e., evalauted $D_a = D_a(\phi_{\text{solver}})$, where ϕ_{solver} is the field values provided by whatever nonlinear solver algorithm is implemented

Sometimes, it is helpful to use both approaches in separate test problems because, together, they isolate different parts of the code.

9.5 Coupled Multi-Physics Problems

Consider a thermal incompressible fluid in index notation where x_2 is the direction of gravity, g:

$$\begin{aligned}
\frac{\partial u_j}{\partial x_j} &= 0 \\
\rho \frac{\partial u_j}{\partial t} + \rho u_k \frac{\partial u_k}{\partial x_k} &= -\frac{\partial p}{\partial x_j} + \mu \frac{\partial^2 u_j}{\partial x_i^2} - \rho g (T - T^{\mathrm{ref}}) \delta_{j2} \\
\rho c_p \frac{\partial T}{\partial t} + \rho u_k \frac{\partial c_p T}{\partial x_k} &= \frac{\partial}{\partial x_j} \left(k \frac{\partial T}{\partial x_j} \right)
\end{aligned}$$

Although it seems quite counterintuitive because there are physics associated with the above equations, it is no less legitimate to add right-hand side functions,

$$\begin{aligned}
\frac{\partial u_j}{\partial x_j} &= M(x_1, x_2, x_3) \\
\rho \frac{\partial u_j}{\partial t} + \rho u_k \frac{\partial u_k}{\partial x_k} &= -\frac{\partial p}{\partial x_j} + \mu \frac{\partial^2 u_j}{\partial x_i^2} - \rho g (T - T^{\mathrm{ref}}) \delta_{j2} + F_j(x_1, x_2, x_3, t) \\
\rho c_p \frac{\partial T}{\partial t} + \rho u_k \frac{\partial c_p T}{\partial x_k} &= \frac{\partial}{\partial x_j} \left(k \frac{\partial T}{\partial x_j} \right) + G(x_1, x_2, x_3, t)
\end{aligned}$$

choose solutions to u_j and T, insert them into the above equations, solve for M, F_j, and G, and apply those to test our solver. Moreover, however, it is important to point out that any of the terms in the above problem statement can be replaced with the chosen, analytical solution. This testing can focus on specific terms, one at a time, while cause all other terms to be evaluated analytically. Then, gradually, groups of terms can be allowed to "float" with the code's solver, with the derived right-hand side forcing functions and boundary condtiions guiding the solution all the while.

Chapter 10

Linear System Solvers

We wish to solve the $N \times N$ linear system

$$\mathbf{Ax} = \mathbf{b} \tag{10.1}$$

using an iterative method, where previous estimates to the solution $\mathbf{x}$ are used to generate improved estimates, and, with each iteration, we come closer to arriving at the actual solution.

10.1 Jacobi Iteration

10.1.1 Basics of the Algorithm

The Jacobi method is among the simplest of methods for solving Equation 10.1. If we let $x^m, m = 1, 2, ..., N$ represent the components of $\mathbf{x}$, similarly for b^m, and A_{ij} represent the ith row and jth column, then Equation 10.1 can also be written as

$$\sum_{j=1}^{N} A_{ij} x^j = b^i \qquad i = 1, 2, 3, ..., N \tag{10.2}$$

We let subscripts denote the iteration number, e.g., $\mathbf{x}_k$ is the kth iterate (guess) for the solution $\mathbf{x}$.

The Jacobi iteration method is this:

1. Given a current guess $\mathbf{x}_k$

2. On each row of Equation 10.1, assume that value is correct for all of the off-diagonal terms, in order to

3. Solve for the diagonal term, making that the next iterate, $\mathbf{x}_{k+1}$

Mathematically, we can write the above sequence of steps as follows. First, let

$$\begin{aligned}
\mathbf{D} &= \text{The diagonal of } \mathbf{A} \\
\mathbf{U} &= \text{The same as } -\mathbf{A} \text{ above the diagonal, otherwise zero.} \\
\mathbf{L} &= \text{The same as } -\mathbf{A} \text{ below the diagonal, otherwise zero.}
\end{aligned}$$

So, on each row, the entries of **U** and **L** represent the off-diagonal coefficients, while **D** represent the diagonal coefficients. To be clear,

$$\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U} \tag{10.3}$$

So, the Jacobi iteration sequence is as follows:

1. Given $\mathbf{x}_k$

2. $\mathbf{x}_{k+1} = \mathbf{D}^{-1}\left(\mathbf{b} + (\mathbf{L}+\mathbf{U})\,\mathbf{x}_k\right]$

3. Continue until some error criteria is satisfied.

10.1.2 Norms and Convergence Criteria

There are multiple ways to measure how close we are to the solution. Here are a few:

- The residual. This is a vector that row-by-row, expresses how different the right and left sides of the equations are for a given iterate, $\mathbf{x}_k$. It is:

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k \tag{10.4}$$

- The approximate error. The error $\mathbf{e}_k$ of the iterate is

$$\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k \tag{10.5}$$

  Of course, to compute it we must know the solution, $\mathbf{x}$, which we typically do not have. We can approximate the error as

$$\hat{\mathbf{e}}_k = \mathbf{x}_{k+1} - \mathbf{x}_k \tag{10.6}$$

Each of the above two metrics are vectors, and so we need some type of norm to reduce the vectors to a scalar in order to make decisions about convergence. Two commonly used norms are

- Maximum norm: This norm is simply the maximum absolute value amonst all of the components of the vector. It is sometimes called the "infinity" norm, and is
$$||\mathbf{x}||_\infty \equiv \max(|x^1|,|x^2|,|x^3|,...,|x^N|,) \tag{10.7}$$
- L-2: This norm is analogous to the magnitude of a vector in Cartesian coordinates, and is
$$||\mathbf{x}||_{L2} \equiv \frac{\sqrt{\sum_{m=1}^{N} x_m^2}}{N} \tag{10.8}$$

One can exercise any or all of these norms to help determine when the Jacobi iterations have converged, and they can be used in combination.

10.1.3 Convergence Analysis

It is helpful to express the algorithm using an "iteration matrix". Looking back at the central equation in the algorithm,

$$\mathbf{x}_{k+1} = \mathbf{D}^{-1}\left(\mathbf{b} + (\mathbf{L}+\mathbf{U})\,\mathbf{x}_k\right] \tag{10.9}$$

we see the latter part of the right-hand side contains a matrix operating on the previous iterate. Disregarding the constant term, i.e. the $\mathbf{D}^{-1}\mathbf{b}$ term, it seems that latter term will dictate which direction the iterate will head. So, let us define the iteration matrix for the Jacobi method to be

$$\mathbf{R}_{\mathrm{Jac}} \equiv \mathbf{D}^{-1}\left(\mathbf{L}+\mathbf{U}\right) \tag{10.10}$$

Then the Jacobi step to form the new iterate may be written as

$$\mathbf{x}_{k+1} = \mathbf{R}_{\mathrm{Jac}}\mathbf{x}_k + \mathbf{D}^{-1}\mathbf{b} \tag{10.11}$$

The exact solution satisfes our original system, which can be written as

$$\left(\mathbf{D}-\mathbf{L}-\mathbf{U}\right)\mathbf{x} = \mathbf{b} \tag{10.12}$$

or

$$\mathbf{D}\mathbf{x} = \mathbf{b} + \left(\mathbf{L}+\mathbf{U}\right)\mathbf{x} \tag{10.13}$$

or,

$$\mathbf{x} = \mathbf{D}^{-1}\left(\mathbf{L}+\mathbf{U}\right)\mathbf{x}+\mathbf{D}^{-1}\mathbf{b} \tag{10.14}$$

In terms of the iteration matrix,

$$\mathbf{x} = \mathbf{R}_{\text{Jac}}\mathbf{x}+\mathbf{D}^{-1}\mathbf{b} \tag{10.15}$$

In other words, the exact solution satisfies the Jacobi iteration statement. Using the above relationship, we can manipulate the Jacobi iteration statement to learn about how the error is affected. Subtract the left-hand side of Equation 10.15 from Equation 10.11, and likewise for the right-hand sides to produce

$$-\mathbf{x}+\mathbf{x}_{k+1} = \mathbf{R}_{\text{Jac}}\mathbf{x}_k+\mathbf{D}^{-1}\mathbf{b}+-\mathbf{R}_{\text{Jac}}\mathbf{x}-\mathbf{D}^{-1}\mathbf{b} \tag{10.16}$$

or, by negating all terms and cancelling the terms involving **b**,

$$\mathbf{x}-\mathbf{x}_{k+1} = \mathbf{R}_{\text{Jac}}\mathbf{x}-\mathbf{R}_{\text{Jac}}\mathbf{x}_k \tag{10.17}$$

or, recalling the definition of the error at iteration k, $\mathbf{e}_k$,

$$\mathbf{e}_{k+1} = \mathbf{R}_{\text{Jac}}\mathbf{e}_k \tag{10.18}$$

So in this relationship, we can see that the iteration matrix affects how the error changes during Jacobi iterations. The eigenvalues of $\mathbf{R}_{\text{J}}$ quantify those effects.

The eigenvectors of a matrix are those vectors for which, when operated on by the matrix are only amplified or diminished in maginutude, and not changed in direction. The eigenvalues are the corresponding amplification factors. Thus, the problem statement that determines the eigenvalues and eigenvectors of a matrix, in our case $\mathbf{R}_{\text{Jac}}$ is

$$\mathbf{R}_{\text{Jac}}\mathbf{w}_i = \lambda^i\mathbf{w}^i \tag{10.19}$$

where λ^i is the eigenvalue and $\mathbf{w}^i$ ar ethe eigenvectors. The eigenvectors are linearly independent and so they form a basis for the vector space occupied by the solution **x**, the error $\mathbf{e}_k$, etc. In particular, the error can be expressed as a linear combination of eigenvectors,

$$\mathbf{e}_k = \sum_{i=1}^{N} c_k^i\mathbf{w}^i \tag{10.20}$$

Substituting that summation into the left-hand side of Equation 10.18 produces

$$\sum_{i=1}^{N} c_{k+1}^i\mathbf{w}^i = \mathbf{R}_{\text{Jac}}\left(\sum_{i=1}^{N} c_k^i\mathbf{w}^i\right) \tag{10.21}$$

or, making use of the eigenvectors,

$$\sum_{i=1}^{N} c_{k+1}^{i}\mathbf{w}^{i} = \sum_{i=1}^{N} c_{k}^{i}\lambda_{k}^{i}\mathbf{w}^{i} \tag{10.22}$$

If all of the eigenvalues are less than one, the error will be shrinking, whereas, if any of the eigenvalues are greater than one, those components of the error will grow with iterations. Therefore, we define the "spectral radius", as

$$\rho\left(\mathbf{R}_{\mathrm{Jac}}\right) \equiv \max(|\lambda^{1}|,|\lambda^{2}|,|\lambda^{3}|,...,|\lambda^{N}|,) \tag{10.23}$$

and use it as an indicator of the convergence rate of the Jacobi iteration method for $\mathbf{Ax} = \mathbf{b}$.

10.2 Gauss-Seidel Iteration

Gauss-Seidel iteration is like Jacobi iteration, the only difference being that in Gauss-Seidel iteration we always use the most udpated values for the components of $\mathbf{x}$ when computing $\mathbf{x}_{k+1}$ The steps of the iteration are as follows:

1. Given a current guess $\mathbf{x}_k$
2. Place those values in temporary storage $\hat{\mathbf{x}}$
3. On each row of Equation 10.1, use the current values in $\hat{\mathbf{x}}$ to determine a new value for the diagonal entry.

Mathematically, we can write the above sequence of steps as follows. First recall the Jacob step from k to $k+1$

$$\mathbf{x}_{k+1} = \mathbf{D}^{-1}\left(\mathbf{b} + \left(\mathbf{L}+\mathbf{U}\right)\mathbf{x}_{k}\right] \tag{10.24}$$

That step can be arrived at by writing the linear system using the L-U decomposition:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \left(\mathbf{D}-\mathbf{L}-\mathbf{U}\right)\mathbf{x} &= \mathbf{b} \\ \mathbf{Dx} &= \mathbf{b}+\left(\mathbf{L}+\mathbf{U}\right)\mathbf{x} \\ \mathbf{x} &= \mathbf{D}^{-1}\mathbf{b}+\mathbf{D}^{-1}\left(\mathbf{L}+\mathbf{U}\right)\mathbf{x} \end{aligned} \tag{10.25}$$

That last line implies the Jacobi iteration scheme:

$$\mathbf{x}_{k+1} = \mathbf{D}^{-1}\mathbf{b} + \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\,\mathbf{x}_k \tag{10.26}$$

In the Gauss-Seidel iteration, we perform a similar sequence, but instead of taking **L** *and* **U** to the right-hand side, we only take **U**:

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{b} \\ (\mathbf{D}-\mathbf{L}-\mathbf{U})\,\mathbf{x} &= \mathbf{b} \\ (\mathbf{D}-\mathbf{L})\mathbf{x} &= \mathbf{b} + (\mathbf{U})\,\mathbf{x} \\ \mathbf{x} &= (\mathbf{D}-\mathbf{L})^{-1}\mathbf{b} + (\mathbf{D}-\mathbf{L})^{-1}\mathbf{U}\mathbf{x} \end{aligned}$$

The second-to-the-last line above is an easier way to write the Gauss-Seidel iteration and the last line reveals the iteration matrix, which can be used for convergence analysis. Using the second-to-the-last line, above, the Gauss-Seidel iteration is

$$(\mathbf{D}-\mathbf{L})\mathbf{x}_{k+1} = \mathbf{b} + (\mathbf{U})\,\mathbf{x}_k \tag{10.27}$$

In summation notation,

$$x^m_{k+1} = \frac{1}{A_{kk}}\left(b^m - \sum_{j=1}^{m-1} A_{mj}x^j_k - \sum_{j=m+1}^{N} A_{mj}x^j_{k+1}\right) \qquad m = 1,2,..,N \tag{10.28}$$

10.3 The Conjugate Gradient Method

The conjugate graidient or “CG” method, was invented by Hestenes and Stiefel, but remained dormant until the advent of digital computers. Here we attempt to provide as concise derivation of the method. The CG algorithm makes use of several ideas simultaneously. It is important to understand these multiple ideas and how they work together in its construction. We begin our derivation by discussing those ideas.

10.3.1 Key Idea #1: The Analogous Minimization Problem

If **A** is symmetric, then the solution to Equation 10.1 is the same as finding the extremum of

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}\mathbf{x} \tag{10.29}$$

In other words, if we denote each component of $\mathbf{x}$ as x_m, then

$$\frac{\partial f(\mathbf{x})}{\partial x_m} = \mathbf{A}\mathbf{x} - \mathbf{b} \tag{10.30}$$

and so when

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial x_m} = 0 \tag{10.31}$$

then $\mathbf{A}\mathbf{x} = \mathbf{b}$. If $\mathbf{A}$ is positive definite, the solution is at a minimum of the quadratic function. If it is negative definite, the solution is at a maximum. Here, we will assume $\mathbf{A}$ is positive definite, hence, we are minimizing f.

The Minimization problem is important because the idea of the steepest descent, which is one method used to find a minimum, plays into the thinking behind the CG algorithm. Consider some current iterate, $\mathbf{x}_i$, and we want to seek the next iterate $\mathbf{x}_{i+1}$. The steepest descent approach to minimization would state that we should seek that next guess by moving in the direction of the gradient of f. In other words, we should move in the direction of

$$\left.\frac{\partial f(\mathbf{x})}{\partial x_m}\right|_{\mathbf{x}_i} \tag{10.32}$$

which, again, is simply this:

$$\mathbf{A}\mathbf{x}_i - \mathbf{b} \tag{10.33}$$

The above expresion is also what we call the “residual”, i.e., a measure of how far off we are from the solution. And so, if we compute

$$\mathbf{r}_i = \mathbf{A}\mathbf{x}_i - \mathbf{b} \tag{10.34}$$

then in the steepest descent approach,

$$\mathbf{x}_{i+1} = \mathbf{x}_i + a_i\mathbf{r}_i \tag{10.35}$$

where a_i is chosen to minimize f along the $\mathbf{r_i}$ direction. As a side note, a_i can be found analytically.

The problem with the steepest descent algorithm is that certain shapes of f can cause very slow convergence, for example, when successive $\mathbf{x}_i$ iterates become stuck crossing back and forth across a narrow valley on the way toward the minimum of f.

Although the steepest descent method is not the focus here, the thinking behind it contributes to the motivation for the CG method. In fact, reflecting on the words "conjugate" and "gradient" provide a clue regarding the motivation: If we could somehow take the steepest descent gradients and use them to create an orthogonal basis for the vector space, perhaps we could use that orthogonal basis to reduce how many back and forths are involved.

10.3.2 Key Idea #2: Creating the Orthogonal Basis from Residuals

The creation of an orthogonal basis is possible, in fact, through a procedure known as Gramm-Schmidt orthogonalization, which allows us to use a set of linearly independent vectors to form a set of orthogonal basis vectors. To make it easier to use the results of this subsection, we will use $\mathbf{r}_i, i = 1,2,3...,N$ as our linearly indpendent vectors and $\mathbf{p}_i, i = 1,2,3...,N$ as the orthogonal basis we wish to derive. The connection between the $\mathbf{r}_i$ here and the residual vectors introduced alrerady is not coincidental. And to be clear, for now we mean orthogonal in the regular sense, with the inner product $(\mathbf{p},\mathbf{r}) = \mathbf{p}^T\mathbf{r}$. The Gramm-Schmidt process is based on the idea of being able to "project" one vector onto another. Projecting $\mathbf{r}$ onto $\mathbf{p}$ is denoted as

$$\text{proj}_{\mathbf{p}}(\mathbf{r}) \equiv (\mathbf{p},\mathbf{r}) \left[\frac{\mathbf{p}}{(\mathbf{p},\mathbf{p})} \right] \tag{10.36}$$

The term in brackets is a unit vector in the direction of $\mathbf{p}$. The leading coefficient is scaled by the amount $\mathbf{p}$ has in common, directionally, with $\mathbf{r}$. So, $\text{proj}_{\mathbf{p}}(\mathbf{r})$ is a vector containing the component of $\mathbf{r}$ that lies in the direction of $\mathbf{p}$.

The process starts with making the first vector $\mathbf{p}_0$ simply equal to $\mathbf{r}_0$. After that, $\mathbf{p}_1$ is taken to be $\mathbf{r}_1$, but we also remove from that the part of $\mathbf{r}_1$ that has components of $\mathbf{p}_0$ in it. Likewise, for $\mathbf{p}_2$, we use $\mathbf{r}_2$ to begin, but subtract away from that the components of the previous results $\mathbf{p}_0$ and $\mathbf{p}_1$ that are contained in

$\mathbf{p}_2$. Thus, the procedure is:

$$\begin{aligned}
\mathbf{p}_1 &= \mathbf{r}_0 \\
\mathbf{p}_2 &= \mathbf{r}_1 - \mathrm{proj}_{\mathbf{p}_1}(\mathbf{r}_1) \\
\mathbf{p}_3 &= \mathbf{r}_2 - \mathrm{proj}_{\mathbf{p}_1}(\mathbf{r}_2) - \mathrm{proj}_{\mathbf{p}_2}(\mathbf{r}_2) \\
\mathbf{p}_4 &= \mathbf{r}_3 - \mathrm{proj}_{\mathbf{p}_1}(\mathbf{r}_3) - \mathrm{proj}_{\mathbf{p}_2}(\mathbf{r}_3) - \mathrm{proj}_{\mathbf{p}_3}(\mathbf{r}_3) \\
&. \\
&. \\
&. \\
\mathbf{p}_{k+1} &= \mathbf{r}_k - \sum_{j=1}^{k} \mathrm{proj}_{\mathbf{p}_j}(\mathbf{r}_k) \\
&= \mathbf{r}_k - \sum_{j=1}^{k} (\mathbf{p_j}, \mathbf{r_k}) \left[\frac{\mathbf{p_j}}{(\mathbf{p_j}, \mathbf{p_j})} \right]
\end{aligned} \tag{10.37}$$

10.3.3 Key Idea #3: A More Convenient Inner Product

Continuing from the above discussion, if $\mathbf{r}_k$ were our residuals, we could use the above process to compute an orthogonal basis. However, it is not convenient to execute the above operation using the standard inner product, $(\mathbf{r}, \mathbf{p}) = \mathbf{r}^T \mathbf{p}$. Rather, and this is a wonderful idea of non-trival value, it is much more convenient to introduce a different inner product,

$$(\mathbf{r}, \mathbf{p})_A = (\mathbf{r}, \mathbf{A}\mathbf{p}) = \mathbf{r}^T \mathbf{A}\mathbf{p} \tag{10.38}$$

Note that because $\mathbf{A}$ is symmetric,

$$(\mathbf{r}, \mathbf{p})_A = \mathbf{r}^T \mathbf{A}\mathbf{p} = (\mathbf{A}\mathbf{p})^T \mathbf{r} = \mathbf{p}^T \mathbf{A}^T \mathbf{r} = \mathbf{p}^T \mathbf{A}\mathbf{r} = (\mathbf{p}, \mathbf{r})_A \tag{10.39}$$

The above definition satisfies all of the requirements of an inner product on a vector space. However, it makes the inner product more relevant to our problem – and more powerful for our problem. In the CG method, we take $\mathbf{r}_i$ as a set of vectors in the space, and assume for now they are linearly independent; we will show later that they are. From them we construct our orthogonal basis set, $\mathbf{p}_i, i = 1, 2, ..., N$ using the Gramm-Schmidt procedure as described above except with our unique inner product. We define our analogous projection operator,

$$\text{Aproj}_{\mathbf{p}}(\mathbf{r}) \equiv (\mathbf{p},\mathbf{r})_A \left[\frac{\mathbf{p}}{(\mathbf{p},\mathbf{p})_A}\right] \tag{10.40}$$

and we seek

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k} \text{Aproj}_{\mathbf{p}_i}(\mathbf{r}_k) \tag{10.41}$$

or, substituting the definition for the A-projection operator,

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k} (\mathbf{p}_i,\mathbf{r}_k)_A \left[\frac{\mathbf{p}_i}{(\mathbf{p}_i,\mathbf{p}_i)_A}\right] \tag{10.42}$$

It is helpful to sequester the scalar in the summation terms, i.e.,

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k} \left[\frac{(\mathbf{p}_i,\mathbf{r}_k)_A}{(\mathbf{p}_i,\mathbf{p}_i)_A}\right] \mathbf{p}_i \tag{10.43}$$

And it is also helpful, at this point, to explicitly write out the **A**-inner product in the numerator and the standard inner product in the denominator

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k} \left[\frac{\mathbf{p}_i^T \mathbf{A} \mathbf{r}_k}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}\right] \mathbf{p}_i \tag{10.44}$$

The bracketed term is a scalar that we refer to as β_{ik}, so that we write

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k} \beta_{ik} \mathbf{p}_i \tag{10.45}$$

where

$$\beta_{ik} = \frac{\mathbf{p}_i^T \mathbf{A} \mathbf{r}_k}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i} \tag{10.46}$$

By virtue of Equation 10.39, it can be written as

$$\beta_{ik} = \frac{\mathbf{r}_k^T \mathbf{A} \mathbf{p}_i}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i} \tag{10.47}$$

10.3.4 Derivation of Rudimentary CG

As discussed above, the CG algorithm is built in part on the notion that we can form an orthogonal basis set $\mathbf{p}_i, i = 1, 2, ..., N$ for the vector space in which our solution $\mathbf{x}$ lies. Building on that assertion, in principle, we can express the solution as a linear combination of that basis as follows

$$\mathbf{x} = \sum_{i=1}^{N} \alpha_i \mathbf{p_i} \tag{10.48}$$

The α_i coefficients are found in the usual way, specifically, we make use of the inner product (in this case, $(\cdot,\cdot)_A$), which will cause all but one term in the summation to be zero. Pre-multiplying each side by $\mathbf{p}_k^T \mathbf{A}$ produces

$$\mathbf{p}_k^T \mathbf{A}\mathbf{x} = \sum_{i=1}^{N} \alpha_i \mathbf{p_k}^T \mathbf{A}\mathbf{p_i} \qquad k = 1, 2, ..., N \tag{10.49}$$

Every term in the summation is zero except for where $i = k$, because the $\mathbf{p}_i$ vectors are all orthogonal in the $\mathbf{A}$-inner product, by design. Hence,

$$\mathbf{p}_k^T \mathbf{A}\mathbf{x} = \alpha_k \mathbf{p_k}^T \mathbf{A}\mathbf{p_k} \tag{10.50}$$

While we do not know $\mathbf{x}$ we certainly know that $\mathbf{A}\mathbf{x} = \mathbf{b}$, so

$$\mathbf{p}_k^T \mathbf{b} = \alpha_k \mathbf{p}_k^T \mathbf{A}\mathbf{p}_k \tag{10.51}$$

Then, solving for α_k,

$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{b}}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k} \tag{10.52}$$

At this point, we can now write what we call here a rudimentary form of CG. It is not practicle for computations, but it has the main theoretical components. The remaining steps in the subsequent section create a computable version of the algorithm. The important thing to remember is that the coming steps do not introduce any new theoretical concepts. Although they are steps critical to the success of CG, and they rely on some of its key ideas, they are only algebraic manipulations. With that stated, the rudimentary CG algorithm is as follows:

$$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 \tag{10.53}$$

$$\mathbf{p}_0 = \mathbf{r}_0 \tag{10.54}$$

From then on:

$$\alpha_{k+1} = \frac{\mathbf{p}_k^T \mathbf{b}}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k} \tag{10.55}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_{k+1}\mathbf{p}_{k+1} \tag{10.56}$$

$$\mathbf{r}_{k+1} = \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1} \tag{10.57}$$

$$\beta_{i,k} = \frac{\mathbf{r}_k^T \mathbf{A}\mathbf{p}_i}{\mathbf{p}_i^T \mathbf{A}\mathbf{p}_i} \tag{10.58}$$

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k} \beta_{ik}\mathbf{p}_k \tag{10.59}$$

$$\tag{10.60}$$

10.3.5 Derivation of Computable CG

The problem with the above procedure is the summation in the equation for $\mathbf{p}_{k+1}$. For very large systems, it is not feasible to store all previous $\mathbf{p}_k$ vectors, and without a simplification of the above algorithm, it would not be useful. Also, we have one lose end. We never proved that the residuals $\mathbf{r}_k$ were linearly independent. In that regard, note that we form $\mathbf{r}_k$ as we go. We will show that if the $\mathbf{r}_k$ are formed as part of the prescribed iterations, above, which includes the Gramm-Schmidt orthogonalization process, the residuals will be linearly independent, and more than that: They will be orthogonal. Also, we will show there is an orthogonality between **r** and **p**. In fact, we will start with that.

Orthogonlity Condition in r and p

To begin, we introduce the idea of the error at iteration k:

$$\mathbf{e}_k \equiv \mathbf{x} - \mathbf{x}_k \tag{10.61}$$

i.e., it is the difference between the current iterate $\mathbf{x}_k$ and the correct solution $\mathbf{x}$. Notice that

$$\mathbf{e}_k = \mathbf{x} - \mathbf{x}_k = \left(\sum_{i=1}^{N} \alpha_i \mathbf{p}_i\right) - \left(\sum_{i=1}^{k} \alpha_i \mathbf{p}_i\right) = \sum_{i=k+1}^{N} \alpha_i \mathbf{p}_i \tag{10.62}$$

Also, the error and residual are related to each other by the matrix $\mathbf{A}$, which transforms error into residual:

$$\begin{aligned}\mathbf{r}_k &= \mathbf{b} - \mathbf{A}\mathbf{x}_k = \mathbf{b} - \mathbf{A}(\mathbf{x} - \mathbf{e}_k) = (\mathbf{b} - \mathbf{A}\mathbf{x}) + \mathbf{A}\mathbf{e}_k \\ &= \mathbf{A}\mathbf{e}_k \end{aligned} \tag{10.63}$$

Returning to the expansion for the error,

$$\mathbf{e}_k = \sum_{i=k+1}^{n} \alpha_i \mathbf{p}_i \tag{10.64}$$

we pre-multiply both sides by $\mathbf{p}_j^T \mathbf{A}$ $(j \leq k)$ to get

$$\mathbf{p}_j^T \mathbf{A}\mathbf{e}_k = \sum_{i=k+1}^{N} \alpha_i \mathbf{p}_j^T \mathbf{A}\mathbf{p}_i \qquad j \leq k \tag{10.65}$$

The left-hand side can be written in terms of the residual, so that

$$\mathbf{p}_j^T \mathbf{r}_k = \sum_{i=k+1}^{n} \alpha_i \mathbf{p}_j^T \mathbf{A}\mathbf{p}_i \qquad j \leq k \tag{10.66}$$

and the right-hand side is zero because each $\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j$ product is zero by design. Note carefully the indices. Here we are letting j range from 1 to k. The lower limit on the summation is $k+1$, which is greater than j. Hence, each term in the summation involves a $\mathbf{p}_k$ vector with an always-previous $\mathbf{p}_i$ value. We enforce in the algorithm, that they are $\mathbf{A}$-orthogonal. So, the above equation is simply

$$\mathbf{p}_j^T \mathbf{r}_k = 0 \qquad j \leq k. \tag{10.67}$$

That is to say, at iteration k, the residual $\mathbf{r}_k$ will be orthogonal (in the regular sense) to all of the previous $\mathbf{p}_j$ vectors. Note again the range on j, it is up to and including k.

Orthogonlity Condition for all r

We now derive a similar statement about the residuals being orthogonal to each other. Returning to the Gram-Schmidt orthogonalization procedure,

$$\mathbf{p}_{k+1} = \mathbf{r}_k + \sum_{i=1}^{k} \beta_{ik} \mathbf{p}_i \tag{10.68}$$

Here, we take the transpose of both sides and then post-multiply by $\mathbf{r}_j$ where, this time, j is greater than k, ($j > k$):

$$\mathbf{p}_{k+1}^T \mathbf{r}_j = \mathbf{r}_k^T \mathbf{r}_j + \sum_{i=1}^{k} \beta_{ik} \mathbf{p}_i^T \mathbf{r}_j \tag{10.69}$$

Examining the terms in the summation, here j is greater than k, meaning that the summation terms involve the product of a newer residual with an older $\mathbf{p}_i$. By virture of Equation 10.67, each of those terms are zero. And on the left-hand side, they are all zero for the same reason. The $k+1$ subscript is why we took special note of the indices in the derivation of Equation 10.67; here, when $j = k+1$, the left-hand side is still zero. See the last sentence of the previous paragraph as a reminder. Thus, we have

$$\mathbf{r}_k^T \mathbf{r}_j = 0 \qquad \text{for} j > k. \tag{10.70}$$

In other words, each new residual $\mathbf{r}_j$ is orthogonal to all previous residuals. The conclusion, therefore, is that enforcing the $\mathbf{A}$-orthogonality of the $\mathbf{p}_i$ vectors results in an orthogonal basis in $\mathbf{r}_i$.

Another Version of α_i

We can use the preceding orthogonality conditions to dervie a different expression for the αs, one that can be used to make a dramatic improvement in the calculation of the βs. We start with Equation 10.70:

$$\mathbf{r}_{k+1}^T \mathbf{r}_k = 0 \tag{10.71}$$

We replace the first residual with the definition of the residual,

$$\left(\mathbf{b} - \mathbf{A}\mathbf{x}_{k+1}\right)^T \mathbf{r}_k = 0 \tag{10.72}$$

Next, use the recursive relationship for $\mathbf{x}_{k+1}$,

$$\left(\mathbf{b} - \mathbf{A}\left(\mathbf{x}_k + \alpha_{k+1}\mathbf{p}_{k+1}\right)\right)^T \mathbf{r}_k = 0 \tag{10.73}$$

Now, rearrange the terms inside the transpose

$$\left[\left(\mathbf{b} - \mathbf{A}\mathbf{x}_k\right) - \alpha_{k+1}\mathbf{A}\mathbf{p}_{k+1}\right]^T \mathbf{r}_k = 0 \tag{10.74}$$

and distribute them

$$[(\mathbf{b}-\mathbf{A}\mathbf{x}_k)]^T\,\mathbf{r}_k - [\alpha_{k+1}\mathbf{A}\mathbf{p}_{k+1}]^T\,\mathbf{r}_k = 0 \tag{10.75}$$

Solve for α_k:

$$\alpha_{k+1} = \frac{(\mathbf{b}-\mathbf{A}\mathbf{x}_k)\mathbf{r}_k}{\mathbf{p}_{k+1}^T\mathbf{A}^T\mathbf{r}_k} \tag{10.76}$$

The numerator is $\mathbf{r}_k^T\mathbf{r}_k$ and in the denominator, $\mathbf{A}^T = \mathbf{A}$, so

$$\alpha_{k+1} = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{p}_{k+1}^T\mathbf{A}\mathbf{r}_k} = 0 \tag{10.77}$$

The denominator can be written using $\mathbf{p}$ only. Recall that

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \sum_{i=1}^{k}\beta_{ik}\mathbf{p}_i \tag{10.78}$$

Rearranging,

$$\mathbf{r}_k = \mathbf{p}_{k+1} + \sum_{i=1}^{k}\beta_{ik}\mathbf{p}_i \tag{10.79}$$

So that the denominator in Equation 10.77 is

$$\begin{aligned}\text{Denominator} &= \mathbf{p}_{k+1}^T\mathbf{A}\mathbf{r}_k \\ &= \mathbf{p}_{k+1}^T\mathbf{A}\left[\mathbf{p}_{k+1} + \sum_{i=1}^{k}\beta_{ik}\mathbf{p}_i\right]\end{aligned}$$

Because $k+1 > i$, each term in the summation of the denominator is zero. Thus

$$\alpha_{k+1} = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{p}_{k+1}^T\mathbf{A}\mathbf{p}_{k+1}} \tag{10.80}$$

A Better Version of β

With the alternative version of α in hand, we can greatly simplify the β calculation and the related summation. We start with the relationship for $\mathbf{x}_{k+1}$,

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_{k+1}\mathbf{p}_{k+1} \tag{10.81}$$

We replace the iterates $\mathbf{x}_k$ $\mathbf{x}_{k+1}$ with expressions involving residuals, in two steps. First, premultiply by $\mathbf{A}$:

$$\mathbf{A}\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \alpha_{k+1}\mathbf{A}\mathbf{p}_{k+1} \tag{10.82}$$

Then subtract $\mathbf{b}$ from both sides,

$$(\mathbf{A}\mathbf{x}_{k+1} - \mathbf{b}) = (\mathbf{A}\mathbf{x}_k - \mathbf{b}) + \alpha_{k+1}\mathbf{A}\mathbf{p}_{k+1} \tag{10.83}$$

Re-write that as

$$-\mathbf{r}_{k+1} = -\mathbf{r}_k + \alpha_{k+1}\mathbf{A}\mathbf{p}_{k+1} \tag{10.84}$$

Now, premultiply by $\mathbf{r}_j^T$, where $j > k$, in other words, premultiply by any of the future residuals. Doing so,

$$-\mathbf{r}_j^T\mathbf{r}_{k+1} = -\mathbf{r}_j^T\mathbf{r}_k + \alpha_{k+1}\mathbf{r}_j^T\mathbf{A}\mathbf{p}_{k+1} \qquad j > k \tag{10.85}$$

The first term on the right-hand side is zero because $j > k$; see Equation 10.70. So we are left with

$$-\mathbf{r}_j^T\mathbf{r}_{k+1} = \alpha_{k+1}\mathbf{r}_j^T\mathbf{A}\mathbf{p}_{k+1} \qquad j > k \tag{10.86}$$

or, rearranging,

$$\alpha_{k+1}\mathbf{r}_j^T\mathbf{A}\mathbf{p}_{k+1} = -\mathbf{r}_j^T\mathbf{r}_{k+1} \qquad j > k \tag{10.87}$$

Recall that

$$\beta_{ik} = \frac{\mathbf{r}_k^T\mathbf{A}\mathbf{p}_i}{\mathbf{p}_i^T\mathbf{A}\mathbf{p}_i} \qquad i = 1,2,...,k \tag{10.88}$$

or using j instead of i,

$$\beta_{jk} = \frac{\mathbf{r}_k^T\mathbf{A}\mathbf{p}_j}{\mathbf{p}_j^T\mathbf{A}\mathbf{p}_j} \qquad j = 1,2,...,k \tag{10.89}$$

Swapping indices k and j,

$$\beta_{kj} = \frac{\mathbf{r}_j^T\mathbf{A}\mathbf{p}_k}{\mathbf{p}_k^T\mathbf{A}\mathbf{p}_k} \qquad k = 1,2,...,j \tag{10.90}$$

Next, we substitute $k+1$ for k,

$$\begin{aligned} \beta_{k+1,j} = \frac{\mathbf{r}_j^T\mathbf{A}\mathbf{p}_{k+1}}{\mathbf{p}_{k+1}^T\mathbf{A}\mathbf{p}_{k+1}} \qquad & k+1 = 1,2,...,j \qquad (10.91) \\ \text{or} \qquad & k = 0,1,..,j-1 \\ \text{or} \qquad & k < j \\ & j > k \end{aligned}$$

Note that the index range on the above version of β is compatible with that in Equation 10.87, so we can use it in that equation. We solve for the numerator in the right-hand side of Equation 10.91,

$$\mathbf{r}_j^T \mathbf{A}\mathbf{p}_{k+1} = \beta_{k+1,j}\mathbf{p}_k^T \mathbf{A}\mathbf{p}_{k+1} \qquad j > k \tag{10.92}$$

and substitute that into Equation 10.87 to get

$$\alpha_{k+1}\left[\beta_{k+1,j}\mathbf{p}_k^T \mathbf{A}\mathbf{p}_{k+1}\right] = -\mathbf{r}_j^T \mathbf{r}_{k+1} \qquad j > k \tag{10.93}$$

Substitute the expression for α_{k+1} that we developed (Equation 10.80) to get

$$\left[\frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_{k+1}^T \mathbf{A}\mathbf{p}_{k+1}}\right]\left[\beta_{k+1,j}\mathbf{p}_k^T \mathbf{A}\mathbf{p}_{k+1}\right] = -\mathbf{r}_j^T \mathbf{r}_{k+1} \qquad j > k \tag{10.94}$$

The above expression has terms in the numerator and denominator that cancel leaving us with

$$\beta_{k+1,j}\mathbf{r}_k^T \mathbf{r}_k = -\mathbf{r}_j^T \mathbf{r}_{k+1} \qquad j > k \tag{10.95}$$

Solving for $\beta_{k+1,j}$,

$$\beta_{k+1,j} = -\frac{\mathbf{r}_j^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k} \qquad j > k \tag{10.96}$$

If we replace $k+1$ with k, we can write

$$\beta_{kj} = -\frac{\mathbf{r}_j^T \mathbf{r}_k}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}} \qquad j > k-1 \quad \text{or} \quad j \geq k \tag{10.97}$$

Notice that the numerator will be zero unless $j = k$. Thus, only one β_{kj} in the entire series summations survive, and that is

$$\beta_{kk} = -\frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}} \tag{10.98}$$

Therefore, we can drop the *kk* subscript and just use *k*, and write Equation 10.59 as

$$\mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k \mathbf{p}_k \tag{10.99}$$

10.4 Multi-Grid

10.4.1 Motivation

In both the Jacobi and Gauss-Seidel iteration schemes, each new entry of the iterate, $\mathbf{x}_{k+1}$ is affected only by the entries of $\mathbf{x}_k$ that are adjacent to it in the mesh. Here, again, $\mathbf{x}$ is the unknown in $\mathbf{Ax} = \mathbf{b}$. For example, our unknown $\mathbf{x}$ could be the values of $\phi_{i,j}$ in a 2-D finite difference grid on which we are solving Laplace's equation. In that case, the discretization would be

$$\frac{\phi_{i-1,j} + \phi_{i+1,j} - 4\phi_{i,j} + \phi_{i,j-1} + \phi_{i+1,j+1}}{h^2} = 0 \tag{10.100}$$

or, since the right-hand side is zero,

$$\phi_{i-1,j} + \phi_{i+1,j} - 4\phi_{i,j} + \phi_{i,j-1} + \phi_{i+1,j+1} = 0 \tag{10.101}$$

The Jacobi iteration can be expressed directly using the $\phi_{i,j}$ values as opposed to using the $\mathbf{Ax} = \mathbf{b}$ notation. First, we solve for ϕ_{ij} in the above discretization,

$$\phi_{i,j} = \frac{\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i+1,j+1}}{4} \tag{10.102}$$

and let that equation be the basis of the iterative method, i.e.,:

1. Given ϕ_{ij}^{OLD} for all grid points,

2. Solve for ϕ_{ij}^{NEW}

$$\phi_{i,j}^{NEW} = \frac{\phi_{i-1,j}^{OLD} + \phi_{i+1,j}^{OLD} + \phi_{i,j-1}^{OLD} + \phi_{i+1,j+1}^{OLD}}{4} \tag{10.103}$$

3. Check for convergence, while also setting $\phi_{i,j}^{OLD} \leftarrow \phi_{i,j}^{NEW}$

4. Repeat until converged

When Jacobi's method is expressed this way, it is easy to see that each grid point value is basically set to the average of the surrounding grid points, over and over, until convergence is acheived. Gauss-Seidel is the same as Jacobi iteration except that the newest values of ϕ_{ij} are always used on the right-hand side. So information propagates a little faster.

But in a broad, high-level way of thinking about information, we can see that information is transmitted only locally in both of these iterative schemes. For example, if information from one point in the mesh needs to be conveyed to another point in a distant part of the mesh, multiple Jacobi or Gauss-Seidel iterations would be required to transmit that information through the interveining points. Conversely, information from one point to the adjacent point is conveyed immediately. For better or worse, that is the nature of the two algorithms.

Let us return now to the $\mathbf{Ax} = \mathbf{b}$ notation, and let us consider fields mapped onto a grid of unknowns, for example, an initial guess $\mathbf{x}^0$ to the system $\mathbf{Ax} = \mathbf{b}$. If there are high-frequency errors in that initial guess, the Gauss-Seidel or Jacobi iterations will dampen them out fairly quickly through the averaging process as shown above. But low-frequency errors, where information must be propagated across large portions of the mesh to be rectified will take longer to dampen.

The fact that errors with frequencies relatively high relative to the mesh are damped out more quickly than those with low frequencies is the main motivation behind multi-grid methods. The idea is to use coarsened versions of the primary grid to eliminate the lower-freqency errors and then map that information onto the more highly-refined mesh, which can be responsible for eliminating the higher-frequency errors.

10.4.2 Demonstration using Two Grid Levels

The simplest demonstration of the multi-grid idea is to consider just two grids, one called F for "fine" and one called C for "coarse", where the grid spacing in C is exactly twice that of F. In other words, every other grid point in F is colocated with a grid point in C, while the remaining F grid points are in-between C grid points.

For clarity, we will pursue a 2-D finite difference simulation of

$$\nabla^2\phi = 0 \tag{10.104}$$

on a square, with uniform gridding in the $x-$ and $y-$ directions. The discretized systems are

$$\mathbf{A}_F\phi_F = \mathbf{b}_F \tag{10.105}$$

for the fine grid, F and

$$\mathbf{A}_C\phi_C = \mathbf{b}_C \tag{10.106}$$

on the coarse grid, C. But, to be consistent with the notation used in linear solvers, we will express these as

$$\mathbf{A}_F \mathbf{x}_F = \mathbf{b}_F \tag{10.107}$$

for the fine grid, F and

$$\mathbf{A}_C \mathbf{x}_C = \mathbf{b}_C \tag{10.108}$$

We begin a simple multi-grid solution by performing a few iterations of Gauss-Seidel on F to obtain $\hat{\mathbf{x}}_F$ We do not expect Gauss-Seidel to converge in those few iterations. But, after they are completed, the residual can be computed on F:

$$\mathbf{r}_F \equiv \mathbf{b}_F - \mathbf{A}_F \hat{\phi}_F \tag{10.109}$$

Recall that the error, $\mathbf{e} \equiv \mathbf{x}_{\text{exact}} - \mathbf{x}_{\text{approx}}$ and residual are related as follows:

$$\begin{aligned}
\mathbf{A}\mathbf{x} &= \mathbf{b} \\
\mathbf{A}\left(\mathbf{e} + \hat{\mathbf{x}}_F\right) &= \mathbf{b} \\
\mathbf{A}\mathbf{e} &= \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}_F \\
\mathbf{A}\mathbf{e} &= \hat{\mathbf{r}}_F
\end{aligned}$$

If we were to solve the above system for $\mathbf{e}$, we could add that value to $\mathbf{x}_F$ and be finished. However, as implied by the previous discussion, $\mathbf{e}$ may have low frequency errors in it that will require many iterations for the Gauss-Seidel solver filter out. So we instead, and this is a key step, map the residual from F onto C, and solve for the error there.

In considering how one would map the residual from F to C, it is tempting to simply take advantage of the fact that every point in C is coincident with a point in F. So, one could simply use that point's value for the residual in F and assign it to the point in C. However, later we will need to map information back from C onto F, and in that case, we will need to perform interpolation for those points in F that do not have a C counterpart.

Because we will need to be doing both mappings, it is helpful to establish some type of symmetric operation that can be used in mapping information back and forth between F and C. To that end, we establish a weighted averaging where each grid point in C receives contributions from all neighboring points in F, weighted according to importance.

For a point p in the coarse grid, there are nine nearby points in F. We identify them as follows:

$$
\begin{aligned}
p_H &\rightarrow \text{The point in } F \text{ coincident with } p \\
p_W &\rightarrow \text{The point in } F \text{ immediately West of } p \\
p_E &\rightarrow \text{The point in } F \text{ immediately East of } p \\
&.. \\
&.. \\
&.. \\
p_{NE} &\rightarrow \text{The point in } F \text{ immediately North and East of } p
\end{aligned}
$$

and so on. The computation of the residual in C at p is weighted as follows:

$$r_C^p = \frac{1}{4} r_F^H + \frac{1}{8}\left(r_F^W + r_F^E + r_F^S + r_F^N\right) + \frac{1}{16}\left(r_F^{NW} + r_F^{NE} + r_F^{SW} + r_F^{SE}\right) \quad (10.110)$$

With $\mathbf{r}_C$ in hand, Gauss-Seidel iterations can be perform to eliminate the lower frequency error, i.e., Gauss-Seidel is applied to

$$\mathbf{A}_C \mathbf{e}_C = \mathbf{r}_C$$

The resulting error on the coarse grid, $\mathbf{e}_C$ is then mapped onto the fine grid using weighted averages as was done above, but they are in the "opposite direction" [1]. Put another way, while in the mapping from F to C we involve all the F points that are immediate neighbors of a C point to contribute to that C point's value, when mapping from C back to F, we allow that C point to influence the values of all of its neighbors. Thus, if we initialize $\mathbf{e}_F = 0$, then the value for each point p in C is added to the immediate neighbors in F in the amounts shown below:

$$
\begin{aligned}
\text{To } e_F^H &\text{ is added } e_C^p \\
\text{To } e_F^W, e_F^E, e_F^S, e_F^N &\text{ is added } \frac{1}{2} e_C^p \\
\text{To } e_F^{NW}, e_F^{NE}, e_F^{SW}, e_F^{SE} &\text{ is added } \frac{1}{8} e_C^p
\end{aligned}
$$

Notice that the amounts are 1 for the coincident point, $1/2$ for the points to the North, South, East, and West, and $1/4$ for the remaining points. Those values, $1, 1/2, 1/4$ are the same relative to each other as the values used in the reverse

[1] and with slightly different scaling, a detail discussed shortly

mapping, $1/4, 1/8, 1/16$. That is not coincidental, and will be discussed in more detail shortly.

But for now, given the $C \rightarrow F$ mapping above, we can form $\mathbf{e}_F$, and use it to correct the result we obtained at the beginning with a few Gauss-Seidel iterations, to produce

$$\hat{\hat{\mathbf{x}}}_F = \hat{\mathbf{x}}_F + \mathbf{e}_F \tag{10.111}$$

With the improved initial guess $\hat{\hat{\mathbf{x}}}_F$, which hopefully has some or all of the lower frequency errors eliminated, we may resume our Gauss-Seidel iterations and expect faster convergence.

10.4.3 Prolongation and Restriction Relationships

In the multi-grid community, the process of mapping a field from the fine grid to the coarse grid is called “restriction”, i.e., it restricts the field on one mesh to one involving approximately half as many points. Likewise, the process of mapping a field from the coarse mesh to the fine mesh is called “prolongation”; it is this process that absolutely must use interpolation.

Consider the prolongation procedure described in the previous section. Notice that, considered as a whole, the process results in various combinations of entries in the coarse mesh to be scaled and added together to produce entries in the fine mesh. It can therefore be written as

$$\mathbf{e}_F = \mathbf{I}_{CF}\mathbf{e}_C \tag{10.112}$$

where $\mathbf{I}_{CF}$ is *not* the identity basis, but rather, the combinations of (1)s, $(1/2)$s, and $(1/4)$s used in the mapping described above. Note that $\mathbf{I}_{CF}$ is $N/2 \times N$

Similarly, the restriction procedure can be written as

$$\mathbf{r}_C = \mathbf{I}_{FC}\mathbf{r}_F \tag{10.113}$$

Note that $\mathbf{I}_{FC}$ is $N \times N/2$. Also, note that

$$\mathbf{I}_{CF} = 4\mathbf{I}_{FC}^T \tag{10.114}$$

10.4.4 The Relationship Between Fine and Course Grid Operators

Since C has exactly twice the grid spacing of F, each of its points are coincident with a point in F. Conversely, every other F point, in the $x-$ or $y-$ directions is in-between two C points. If h is the grid spacing in F, then $2h$ is the grid spacing in C. Similarly, if n_c is the number of divisions in the $x-$ and $y-$ directions on F, then $n_c/2$ is the number of divisions in those directions on C. Note that the number of *points* in the $x-$ and $y-$ directions are $n = n_c + 1$. Both F and C are a logical $i - j$ grid, each with its own, independent $i - j$ numbering. To fix ideas, F's point identified by $(i, j) = (3,5)$ corresponds to C's point $(2,3)$. In addition to the (i, j) method of identifying points, we implement what is sometimes called the "natural ordering" of point numbers, which start with 1 in the lower left corner and are incremented by one in the horizontal direction, then starting with the next row up, are incremented by one, and so on, in other words, the point number on the fine grid is p_F

$$p_F = i_F + (j_F - 1)(n+1) \tag{10.115}$$

and on C,

$$p_C = i_C + (j_C - 1)(n/2+1) \tag{10.116}$$

Now, let us examine the finite difference discretization of Poisson's equation on C, at a point (i_C, j_C):

$$\frac{\phi_{i_C-1,j_C} - 2\phi_{i_1,j_C} - \phi_{i_C-1,j_C}}{(2h)^2} + \frac{\phi_{i_C,j_C-1} - 2\phi_{i_C,j_C} - \phi_{i_C,j_C+1}}{(2h)^2} = f_{i_c,j_c} \tag{10.117}$$

and on F at a point (i_F, j_F):

$$\frac{\phi_{i_F-1,j_F} - 2\phi_{i_1,j_F} - \phi_{i_F-1,j_F}}{h^2} + \frac{\phi_{i_F,j_F-1} - 2\phi_{i_F,j_F} - \phi_{i_F,j_F+1}}{h^2} = f_{i_F,j_F} \tag{10.118}$$

These two equations are very similar. Let us pull the 4 out of the demoninator of the C version:

$$\frac{1}{4}\left[\frac{\phi_{i_C-1,j_C} - 2\phi_{i_1,j_C} - \phi_{i_C-1,j_C}}{h^2}\right] + \frac{1}{4}\left[\frac{\phi_{i_C,j_C-1} - 2\phi_{i_C,j_C} - \phi_{i_C,j_C+1}}{h^2}\right] = f_{i_c,j_c} \tag{10.119}$$

By comparing the above two equations, we see that the operator on the discrete field ϕ_{i_C,j_C} is identical to the operator on the discrete field ϕ_{i_F,j_F}, with the exception of the 1/4 scaling factor. Thus, row p_C in $\mathbf{A}^C$ is identical to the corresponding row p_F in $\mathbf{A}^F$ times 4. That same factor of 4 appears as a scaling factor in Equation 10.114.

In fact, it turns out that for some discrete schemes, it can be shown that

$$\mathbf{A}_C = \mathbf{I}_{CF}\mathbf{A}_F\mathbf{I}_{\mathbf{FC}} \tag{10.120}$$

or

$$\mathbf{A}_C = \alpha \mathbf{I}_{FC}^T \mathbf{A}_F \mathbf{I}_{\mathbf{FC}} \tag{10.121}$$

where, here $\alpha = 4$. Thus, the symmetry of the restriction and prolongation operation results in a way to construct the coarse grid operator from the fine grid operator.

10.4.5 Algebraic Multi-Grid

In algebraic multi-grid, the linear solver is not given the mesh, nor the discretization method used to form the linear system. It is only given **A** and **b**. Algorithms are used to detect, from **A**, what the connectivity between associated vertices are. From that association, algorithms infer which points are on the fine mesh and which are on the coarse mesh.

So, one essential requirement of an algebraic multi-grid ("AMG") method is to determine, approximately, which nodes (unkonwns in **x**, i.e., vertices in the mesh) are only part of the fine mesh, i.e., "*F* nodes", which should also correspond to nodes on the coarse mesh, i.e., *C* nodes. [6] [5]. To do that, first we introduce a way to determine if a node *i* is "strongly coupled" to another node *j*. One way to do that is to examine row *i* of **A**, looking at each of the off-diagonal entries in that row. All of the off-diagonal entries that are at least about 1/4 as big as the largest entry on that row (which is probably on the diagonal), represents a node that is "strongly coupled" to *i*. The idea is that for most PDEs, nodes on row *i* that are physically close to node *i* will have bigger entries than nodes physically farther away. The determination of which nodes are *C* nodes uses the idea of strong coupling as defined through that process.

We start, arbitrarily, by declaring one particular node a *C* node. For the rest of the nodes, *i*, in the mesh, an importance factor, λ_i is computed as follows:

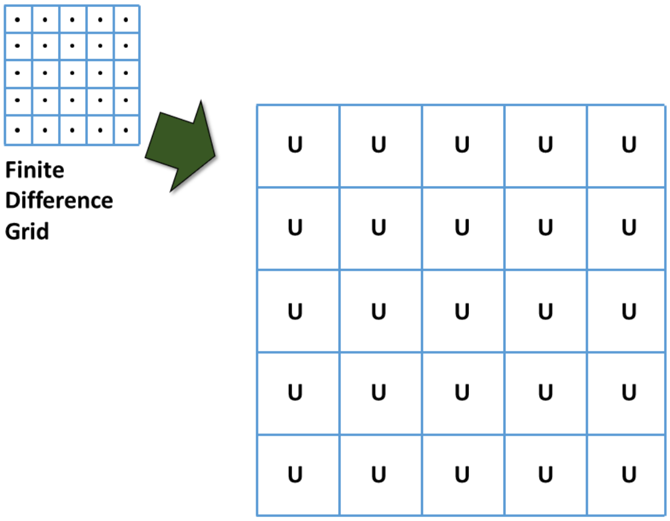
$$\lambda_i \equiv U_{\text{SC}} + 2F_{\text{SC}} \tag{10.122}$$

where

$U_{\text{SC}} =$ the number of still "uncertain" nodes that are "stongly coupled" to *i* and,

$F_{\text{SC}} =$ the number of *F* nodes that are strongly coupled to *i*

While the AMG solver does not have access to the grid and discretization method that gave rise to the matrix upon which it operates, we can still use the grid to examine how the above algorithm peforms for a 2-D finite difference calculation.

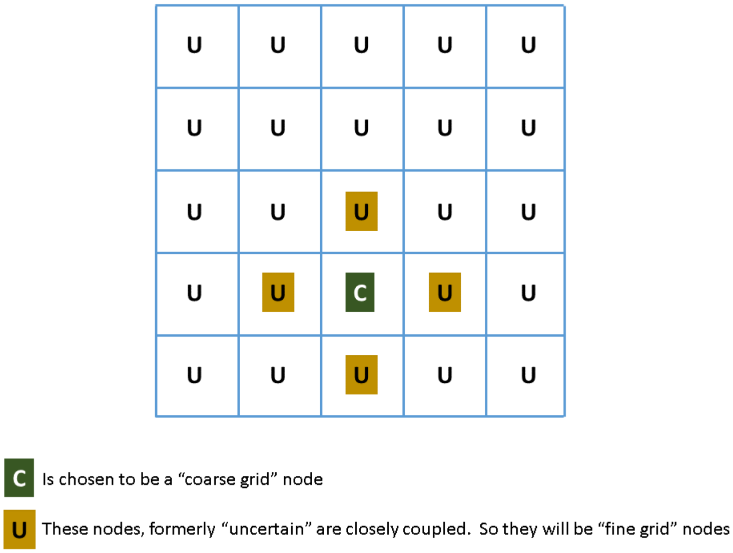


U	U	U	U	U
U	U	U	U	U
U	U	U	U	U
U	U	U	U	U
U	U	U	U	U

Figure 10.1: The finite difference grid, amplified showing that, initially, all nodes are uncertain regarding whether they are coarse-grid or fine-grid nodes.

We start with a grid where every node is labled as "uncertain", "U"; see Figures 10.1 and 10.2. Figures 10.3 - 10.8 show the progression of the algorithm.

Finally, there are other topics and aspects to multi-grid and AMG that are important, at least to know they exist. There is a smoothing step involved AMG that is performed before mapping fields to coarser grids. Also, it is important to note that there are far more grid levels that can be involved in multi-grid methods, using much coarser grids, performing "V" cycles, and even "W" cycles.

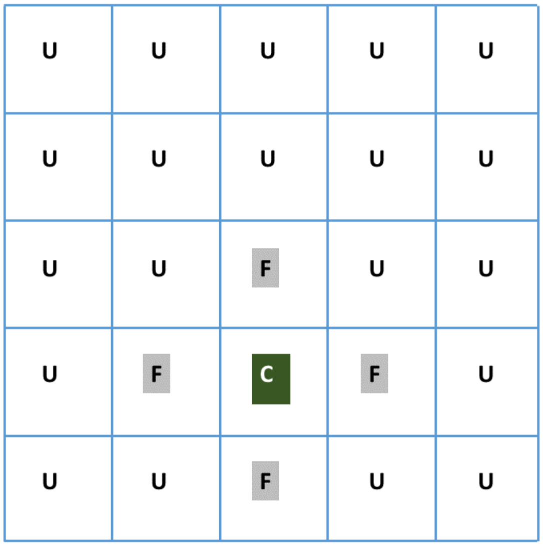


U	U	U	U	U
U	U	U	U	U
U	U	U	U	U
U	U	C	U	U
U	U	U	U	U

C Is chosen to be a “coarse grid” node

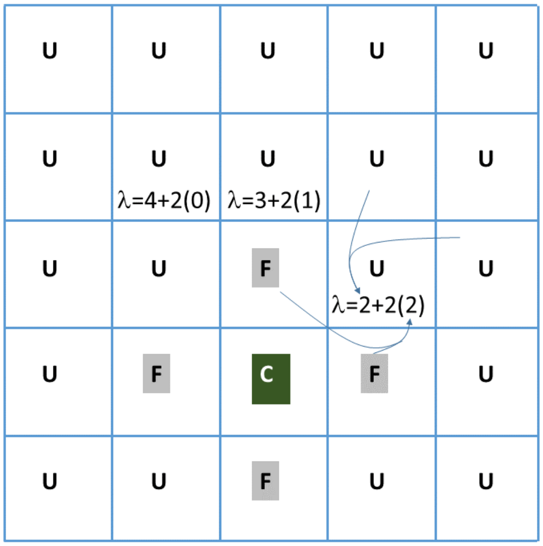
U These nodes, formerly “uncertain” are closely coupled. So they will be “fine grid” nodes

Figure 10.2: A node is chosen as a coarse-grid node arbitrarily. The closely coupled nodes are identified, as shown in gold boxes here, and will be labeled as fine-grid nodes (see next figure).



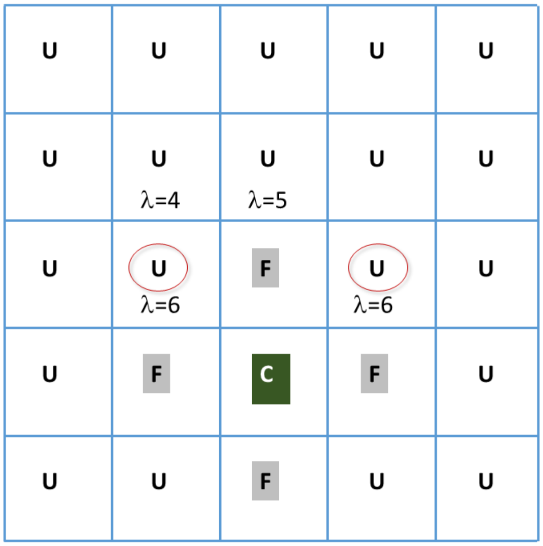
U	U	U	U	U
U	U	U	U	U
U	U	F	U	U
U	F	C	F	U
U	U	F	U	U

Figure 10.3: The fine grid nodes identified from previous step are tagged.



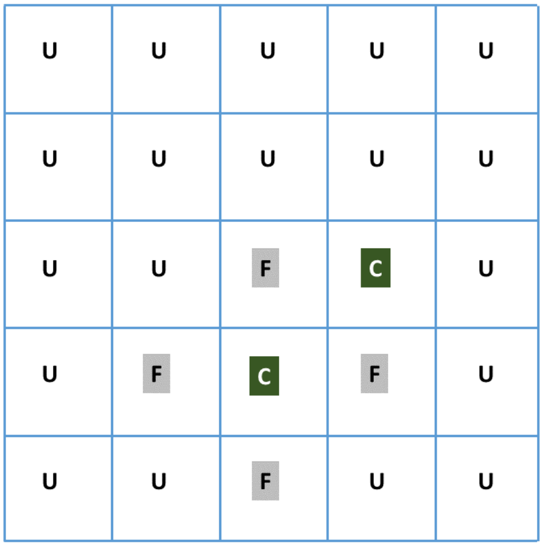
U	U	U	U	U
U	U λ=4+2(0)	U λ=3+2(1)	U	U
U	U	F	U λ=2+2(2)	U
U	F	C	F	U
U	U	F	U	U

Figure 10.4: The λ value is computed for the uncertain nodes. The λ values for a few cells are shown here, with lines indicating from where the values originate.



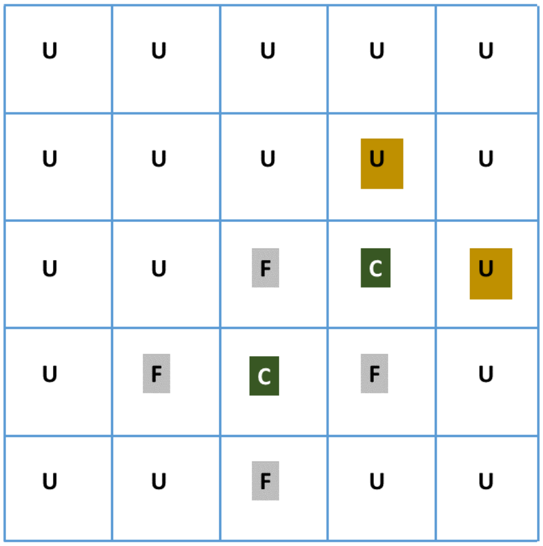
U	U	U	U	U
U	U λ=4	U λ=5	U	U
U	U λ=6	F	U λ=6	U
U	F	C	F	U
U	U	F	U	U

Figure 10.5: The highest values of λ are identified. One will be picked as the next coarse-grid node.



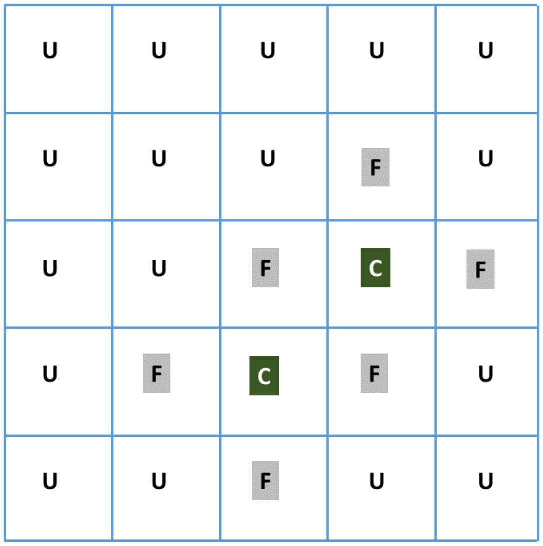
U	U	U	U	U
U	U	U	U	U
U	U	F	C	U
U	F	C	F	U
U	U	F	U	U

Figure 10.6: Here the next coarse grid node is tagged.



U	U	U	U	U
U	U	U	U	U
U	U	F	C	U
U	F	C	F	U
U	U	F	U	U

Figure 10.7: The nodes closely coupled to the new coarse-grid node will become fine grid nodes.



U	U	U	U	U
U	U	U	F	U
U	U	F	C	F
U	F	C	F	U
U	U	F	U	U

Figure 10.8: Those new fine grid nodes are tagged. Notice that the coarse-grid version of the 5-point stencil is emerging from this algorithm.

Chapter 11

Software Testing for Computational Physics Software

11.1 Types of Test

It is highly valuable to test software as it is being developed and when it goes into production. Tests provide evidence that the software is functioning as designed, and this evidence is more than just vague notions of quality. Tests should provide quantitative data for the performance of the code in multiple ways, including accuracy on particular problems, rates of convergence, and execution time. In this section, we discuss in more detail some ofthe the types of tests that should be performed on production code.

11.1.1 Verification

A verification test is one in which there is an answer provided that the software is supposed to match in some sense. Verification test problems can include, for example:

- **Application Domain Analytical Solutions:** These are test problems that have analytical closed form, or semi-analytical solutions that are in the domain of the the code's application. For example, if one is writing a solver for Maxwell's equations, an analytical solution to those equations for homogeneous material in a simple domai exists. Or, if one is writing a hydro-code, there are a handful of analytical solutions for test cases that have some physical significant. The same is true for heat conduction and viscous fluid flow.

These analytical and semi-analytical test cases have in common the idea that they are analytical solutions for simplified *applications* that are related to, in some sense, the actual application.

- **Manufactured Solutions:** These are also test cases that have analytical solutions, however, they need not have any physical relevance. Rather, they are artificial in nature. Still, they have an analytical solution against which the performance of the code can be performed.

- **Unit Analytical:** These types of test problems typically test a very small part of the code, e.g., a subroutine or small set of subroutines, or a particular capability by itself. A unit analytical test case solution need not be a mathematical equation, although it can be. It might be used to test the performance of a heuristic or phenonenological model. The "analytical" or "by-hand" result may consider if-then conditionals and other aspects of the heuristics assocaited with the subroutine or sets of subroutines used.

11.1.2 Validation

A validation test is one that demonstrates the algorithm implemented is effective for the intended purpose. In practice, validation often is synonymous with comparing a code's output to experimental data. In short, it is often said that "Verification is the process of ensuring we implemented the equations correctly while validation is the process of ensure we implemented the correct equations in the first place." Clearly, however, there is more to codes and methods than equations. Still, the idea is sound. And while validation usually means comparison to experimental data, it can also mean on a smaller scale determining if the method implemented is effective in accomplishing its purpose.

Validation testing is often part of a larger effort because, again in practice, it has to do with comparison to experimental data. An experiment for which data is collected is set up and run in the code. The output from the code is compared to the data from the experiment. In a rudiementry sense, if they match, the code is validated. If they do not, it is not validated.

However, there are many subtleties to those ideas. There is the concept of experimental error, numerical uncertainty, grid convergence, etc. The process of validation is a discipline that requires rigorous attention to all of the factors going into code-experiment comparison. And, validation can be performed on the small scale, which has a different connotation altogether.

For example, consider the development of a strength model for a material, perhaps one that involves a handful of adjustable parameters. These parameters are determined experimentally by deforming the material under controlled circumstances in the laboratory, measuring the material's response to stimulus. The model is fit to the data from that experiment and, in a very real sense, is validated in that process. It can be that a model fails to match experimental data in that phase of development, and in that case, it is unlikely it will be implemented in a computational physics code. But if it *is* validated, i.e., if it appears to do well in matching experimental data in the laboratory, it may be implemented in an multi-physics code. In that case, the process of validating that model takes on new meaning. In one sense, the validity of the model has been established and all that is needed is to verify that the model is implemented correctly in the host code. In so doing, it is likely that the experimental data used for the original model validation will be used for comparison. Doing so helps verify the model is implemented correctly, all the parameters and units are understood, and adequate communication has occurred between the experimentalists, theoreticians, and code developers. So, in that case, experimental data is used to verify the model.

However, in another sense, more validation can still occur. The experimental setup used to provide validation data for the model might be very limited. The true test of the model might ultimately be in the actual application for which it was developed. In this case, the validation work mentioned above can be considered a "Small Effects Tests" version of validation. It does not validate the model's interaction with the other more involved and complex physics of the host code. Thus, more elaborate applications-level experiments and the simulation results for them can be compared to further validate the model.

11.2 Regression Testing

Regression testing is an essential feature of any computational physics software development endeavor. Whether it be a research code or a full-scale production code, regression testing is a key element to ensure developers know where things stand with the code as it changes and the platforms on which it runs changes.

The purpose of a regression test is to ensure that a capability that has been tested and, in a sense, established to be working in a code is not broken or damaged by future changes. All of the verification and validation tests mentioned above can serve as regression tests. Ultimately, the goal is to test all capabilities in the code and have each of those capabilities somehow addressed in one or more of the regression test suite.

A regression test system must be implemented that manages the process of running each regression test and reporting on the results. Typical features of a regression test system include the following:

1. **Automated Test Execution** The system must be able to execute a single test case automatically and generate results for that test.

2. **False Passing Prevention** The system must be designed to prevent false positives regarding pass results. In other words, care must be taken to ensure that a result of "passing" is legitimate. It is easy to overlook various factors that might cause the regression system to produce a false "passing" result for a test. For example, if the executable failed to compile correctly and simply does not exist, and if old result files from previous runs still do exist, the test system could try to run the executable, fail at doing that, but then move forward on comparing files, old files that is, to get a passing result that is not deserved.

3. **Ease of Difference Examination** A good regression test system should allow for the user to easily examine differences between the code's current results and the accepted "gold standard" results. Easy comparison helps the developer determine if the differences between the new results and the previously accepted results represent an improvement to the code's capability or the loss of a capability in the code.

4. **Flexible Comparators** For a complex and flexible computational multiphysics code, the regression test system should have the ability to use different methods for comparing results. For some results, the comparison of an $x-y$ plot to the gold standard might be sufficient. And it might be sufficient for that comparison to be digit by digit, in other words, an exact match is required for passing. In other situations, though, it may be allowable for the two plots to differ within a tolerance, e.g.,10^{10} to account for different computer platforms on which the code is tested.

5. **Report Summary** An "all clear" result or any failures should be easily identifiable by the developer. The regression system must provide report summaries for each individual test and also for the entire suite.

11.2.1 Code and Capability Coverage

Automated systems exist to allow one to determine which parts of the source code have been executed by a particular test case. “Code coverage” is often considered the percentage of the lines of code that have been executed, collectively, by the entire regression suite. Although this can be a misleading metric, it is still often used and because there are automated systems that can measure it, it provides a valuable metric. Again, however, there are subtleties to this metric that must be understood in a complex multi-physics situation.

One aspect that should be understood is that the applications for which the code is run might exercise parts of the code that are not tested. Even if the regression tests cover 95% of the code’s executable lines, an application might use that remaining 5% very heavily, and in fact, that remaining 5% could constitute nearly all of the science-related executable lines for that application. The idea of “application code coverage” is introduced to handle this possibility. Application code coverage is the percentage of code executed by an application that has been executed during the regression suite. Automated systems have been developed to measure application code coverage as well.

However, there is still more to consider, and to do so, it is helpful to decrease the granularity in the discussion of code coverage. We intrudoce the idea of “Capability Coverage”. Instead of considering only numbers of lines of code executed by tests, or the numbers of lines of code executed in applications that are also executed in tests, we group blocks of code together that represent a particular capability. One can imagine the following capabilities, some of which are hierarchical, in a simple finite element diffusion solver:

1. Input parser
2. Output file writer
3. Plotter
4. Explicit time marching algorithm
5. Implicit time marching algorithm
6. Steady state solver
7. Non-linear solver
8. Element matrix computation

9. Material model for thermal conductivity
10. Material model for specific heat
11. Material model for density
12. Shape function generator
13. Gauss quadrature routines
14. Matrix assembly
15. Fixed boundary condition application
16. Flux boundary condition application
17. Other custom boundary condition application
18. Matrix solver

This is not an exhaustive list, but it provides an idea about how blocks of executable lines, together, represent different capabilities in the code. And, again, some are hierarchical. The element matrix computation requires the material models for thermal conductivity, specific heat, and density. But, still, the code for those material models can be thought of as capabilities. Certainly, one could write a code that uses constant or hard-coded values. But more sophisticated models could also be implemented, including tabular look-ups and algebraic curve fits. Errors, both in input parsing, interpolation, and otherwise can creep in for these more complex capabilities with respect to the material models. Identifying them as an item, in their own right, as something worthy of testing brings significant value to the regression test process.

With the idea of capabilities having been identified, one can imagine that one goal of the regression suite would be to ensure all capabilities are tested at some point. However, grouping them as capabililties offers the advantage that we can speak now, more easily, of capability combinations. We can write a regression test that exercises many of the capabilities above. However, unless the test is very complex, it is difficult to include them all. And, furthermore, the more capabilities are included, the harder it is to verify the results. It is easier, from a verification standpoint, to run simple problems with fewer capabilities beikjng exercised.

As a result, it is likely that we have a suite of tests, each verifying separate capabilities, and some verifying groups of capabilities. It is helpful, therefore, to keep track of what combinations of capabilities have been verified in a regression test. When running the application, the capabilities and their combinations can be tracked and compared to what is tested in the regression test system.

11.2.2 Test Quality and Application Confidence

Not all regression tests represent the same rigor and value. Some tests may represent rigorous comparison to analytical solutions, small-scale validation data, or large-scale integrated validation tests. Other tests, may simply represent results that seem right, and seem to capture the intention of the code, but do not correspond to by-hand calculations. It is therefore valuable to provide a rudimentry evaluation of the quality of tests so that when the question is asked regarding which capabilities used in application have been tested, a more meaningful answer can be provided. It is worthwhile to consider a simple ranking system such as:

- **High Rigor** Tests in this category could carry with it the highest rigor score of 3 out of 3. These tests are ones that involve by-hand calculations or some type of highly-detailed analysis that shows the code is performing exactly as expected on all of the capabilities involved in the test.

- **Medium Rigor** Tests in this category could carry with it the middle rigor score of 2. These types of tests could contain, for example, trend studies where multiple runs are combined where, in each run, a parameter is varied. The code produces the expected trend in results and is therefore verified. For example, it could produce the correct rate of spatial convergence on a manufactured solution.

- **Low Rigor** Tests in this category could carry a score of 1, the lowest value but not entirely without value. Tests like these only capture what the developer vaguely expects the code should do. These types of tests often represent the capturing of capability, not the verification of capability.

When establishing a regression test, using the above rating guide, it can be given a score of 1, 2, or 3. When that test is run on the code, and the capabilities executed in that test are recorded, the rating of the testing for those capabilities is established. The same is true for the combinations. Then, when an application is run, the capabilities it uses along with the combinations is uses is recorded and projected against those tested. In so doing, it is possible to generate reports on the confidence in the simulation.

Moreover, it is valuable to record at what time in the simulation a capability is first used. We expect, in general, for more capabilities to be used as the simulation progresses. As more capabilities become active, they are increasing in greater and greater numbers the combinations of capabilities in play. The liklihood that all combinations have been tested rigorously decreases as simulation time progresses. So tracking the rigor of testing of the combinations used, over time, can be used to establish a metric of confidence over time. It can also be used to greatly accelerated debugging of the code. A code crashing or producing spurious results can sometimes be correlated with a new capability coming into play. If the testing and capability coverage system is in place, it is possible to more quickly pinpoint that occurrence and lead the developer to the right spot in the code immediately.

11.3 Test Harness

It is far easier to test a code if it is developed with testing in mind. While it is possible to create a test harness for an existing code, doing so requires non-trival experience in the code and experience in developing test infrastructure in general. In this course, we will discuss ways to create a test harness for a code as it is developed. With that knowledge in hand, students who eventually work on large pre-existing codes will be able to contribute productively to adding or using their test harnesses.

A test harness is a way to run small, individual parts of the code either by themselves or in groups while providing inputs that go directly to those parts, and examining the results of those parts. Imagine a particular subroutine in a very large code that requires testing. And imagine that subroutine is not called from the main program, or even one level down from the main program. Instead, it is buried 5, 6, 8, or more levels down in the execution tree. The purpose of the test harness is to enable the testing of that subroutine using inputs provided from an input file. While, under normal operations, the subroutine may largely use values provided by other parts of the code in combination perhaps with some user inputs, under testing it may be driven solely by inputs read from a file. Or, if a small group of subroutines are being tested, they might exchange information in addition to using information from a file. Sometimes, subroutines access very large data structures that are constructed by different parts of the code and it is desirable to not have those other parts active in the test. In those cases, it is necessary for the test harness to supply those data structures. Clearly, therefore, adding test harnesses to some pre-existing codes might be quite challenging, indeed. However, if the possibility of creating a test harness for each subroutine or function is a *design requirement*, then the ability to test the code evolves naturally with the code itself.

There are multiple ways to introduce a test harness system into a code. One way is to allow the code to execute normally while also maintaining a list of routines that are to be tested and those that are to be explicitly skipped during testing. In this approach, the code executes normally until it encounters a routine that is the list of routines to be tested. The fact that a routine is to be tested is detected by a small code snipped inserted at the beginning of the routine. If the code execution is proceeding normally and this code snippet indicates testing is to occur, the snippet calls the test harness for that routine. The test harness sets whatever values are required for the succesful test of the routine and then calls the routine again. When the test harness calls the original routine to be tested, it must pass or set a flag indicating that the code is now in "test mode". Thus, this second time the code snippet at the beginning of the subject code is encountered, it knows to proceed. Once the routine finishes, the test harness may do whatever is necessary to report or evaluate the results. When the test harness is finished with its task, it returns control to the snippet at the top of the subject routine, which now proceeds in normal operation.

The process of determining if the code is to be tested and the process of setting values to be used in the test requires additional data structures that carry information from the user, including the list of routines being tested and the data for each of those routines. In addition to the list of routines being tested, it is possible to create a list of routines to be skipped when the code is in testing mode.

11.4 Example Regression Test System

Shown in Figure **??** is the layout of a simple regression test system. The components of the system are described below. First, the system components that must reside in each test case directory are described. After that, the system components that reside, together, in a directory reserved for the test system, the "regression test system" directory, are described.

Components in each test directory

- *clean.py* This script must exist in each test directory. It is responsible for ensuring old results are removed before a test is executed. If this script malfunctions such that old results that passed regression remain and can pollute future results. In particular, if a new test is run and if that new test fails catestrophically, the old results if they remain may lead the test system to conclude the regression passed. Because each test may involve different types of files for comparison, this file is customized for each test.

- *compare.py* Each test must have a script by this name its directory. Its purpose is to compare the new results, stored in the test case directory, to the "gold standards" files also stored in that same directory. Each test may require a different type of comparison, and so this script may be different in each test case. In the example provided here, each instance of "compare.py" makes reference to a standard, common comparator called "strict_comparator.py", which is in the Python directory that stores the regression test system.

Components in the regression test system directory

- *run_case.py* The responsibility of this script is to execute a singe test case, including running the executable and running the comparator. In the example provided here, the *run_case.py* script moves into the test case, executes the code to be tested, and runs the comparator. Note that the comparator is responsible for writing the *TestResults_Summary* file for the test, and that test resides in the test case directory for later use.

- *run_all.py* This script is to be run in the directory that is the parent of the *Cases* directory. It is responsible for running the *run_case.py* script for each test case in the suite of regression tests. It is then responsible for reading each of the *TestResults_Summary* file in each of those test case directories and compiling the results in a *TestResults_Summary* file in a higher-level directory, from which it is run.

Chapter 12

Integrated Software Development and Documentation Planning for Computational Multi-Physics

The nature of computational physics lends itself to a focus on the complexities of the science, mathematics, and hardware issues that confront the developer because these aspects, in and of themselves, are very challenging. It is beyond most people if not everyone to have a full grasp of all the issues assocaited with every aspect of the most physically diverse and advanced computational software package. And because the development of the software is equally complex and demanding, it seems necessary to devote all of one's intellect to mastering and manipulating the various components of the field to produce a working code.

But in fact, it only *seems* that way. The reality is that, while computational physics is demanding, it is no more demanding than building highly complex, carefully engineered scientific hardware intended for scientific discovery. The concepts behind high-precision equipment such as space telescopes, scanning electron microscopes, and sensors of all sizes and sorts is also demanding and requires the integration of multiple fields amidst changing requirements. But the individuals and organizations that produce such products must, out of necessity, adopt an engineering mindset. Without doing so, they cannot be successful, or at cannot beat the competition who does adopt that mindset to produce a working product.

Adopting a engineering based, planning, mentality has many signficant benefits. The goals are:

1. Reducing Stress: Good planning not only reduces the stress for the individual developer or team, it also decreases the financial stress associated with the project and the stress in business relationships, e.g., between the development team and the sponsor.

2. Demonstrating Due Diligence: As a professional computational physics software developer developing and tracking progress on a development plan is doable. Because it is doable and because it provides such great value, it is representitive of due diligence. It is not only reasonable and appropriate to develop and track a plan, it demonstrates transparency, responsibility, and accountability.

3. Tracking Progress: The ability to track one's progress against a plan demonstrates success on the many small tasks associated with implementing software. It is very tempting to become discouraged when a bug or other problem is making it impossible to create that great result sought by the developer, and probably the sponsor. If one does not track progress in a way that demonstrates success on the smaller tasks, one may feel as though they have absolutely nothing to report. There is no way for the sponsor to know if anything good is being done or if the money is being wasted. Showing progress through every required task develops trust between the sponsor and development team

4. Communicating Issues: When the plan and progress on that plan are tracked, then when an issue does arise the sponsor can be informed so that resources can be adjusted appropriately. Perhaps more money is needed, or more staff, or simply more time. Engaging the sponsor on issues with a plan as the backdrop brings predictability into the picture.

5. Building a Body of Work: While the process of actually writing code is the essential ingredient to the software development process, it is a mistake to think it is the only ingredient. Testing and documentation, evolved and developed as the code implementation progresses is a valuable way to build the body of work in a tangible way that can be appreciated by the sponsor, and by the development team.

One need only think about the sleepless nights and high-pressure environment assocaited with the situation where there is a bug in the code, and the code is not producing the right results. Without a plan, without an ever-evolving set of documentation and tests, there is nothing to show for own's work. It is far better to be able to demonstrate a history of steady progress through design stages, low level testing (hopefully many in number), and increasing documentation. When the developer or team is "stumped" on a problem, that aspect of being stumped is not the only data that the sponsor has to evaluate the developer(s). And when the bug is finally found and the great results are being produced, that aspect is also not necessarily the saving element for the situation. Sponsors need to understand how things came to be, and need more information than the transition from "We have nothing to show" to "We're practically finished." The planning aspects and the mentality of building a body of work greatly assists everyone in making accurate assessments and building trust in good times (great results) and bad (bugs).

12.1 The Development Cycle and Building a Body of Work

The process of developing computational physics code, whether it be an additional capability for an existing code, or a new code altogether, has as its predecessor theory, background discussion, and an identified need. Oftentimes, requirements, meetings, and plans are developed in a cycle, gradually being refined. Once the funding has been allocated for the development, it is greatly beneficial for the development to be done in some type of cyclical manner. One example of a cyclical development is shown in Figure **??**.

The process can begin with writing a small portion of the code, and testing that code on some input values. The output of the code is verified and the cycle begins a second loop. On the second loop, perhaps more input capability is added and verified. Eventually, some functionality is added to the code. When a functionality is added, it should be tested. Using input values and a test harness, the output of the functionality is verified. If it is non-trivial functionality, it is likely that the verification should constitute a unit test in the regression suite. The cycle continues with more capability being added, each time being verified in the cycle, and all the while, ensuring the regression suite passes, or is modified if necessary.

Eventually, capability significant enough is worthy of being included in a report. Automated systems should be set up as part of the regression test system to generate plots and move those plots to the report section of the repository. In this way, with each running of the regression tests, the plots in the report are updated. And with each signficant support capability towards the end goal, the report grows in volume and thoroughness. In this way, the software development process is the building of a body of work that a sponsor can review and in which they can take confidence, and build trust in the development team.

Bibliography

[1] Tannehill John C. Andseron, Dale A. and Richard H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Hemisphere Publishing Corporation, New York, 1984.

[2] Carey Graham F. Oden J. Tinsley Becker, Eric B. *Finite Elements an Introduction*. Prentiss-Hall, International, 1981.

[3] I. G. Currie. *Fundamental Mechanics of Lufids*. McGraw-Hill Book Company, 1074.

[4] Doron Levy. *Numerical Methods for Linear PDEs*. AMSC/CMSC Course 661, 2013.

[5] Luke N. Olson. *Algebraic Multigrid Methods*. Springer Berlin Heidelberg, 2015.

[6] K. Stuben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128:281–309, 2001.