Appendix S

Mathematica Realization of TPSA and Taylor Map Computation

S.1 Background

The forward integration method (Section 10.12.4) for computing Taylor maps can be implemented by a code employing the tools of *automatic differentiation* (AD) described by Neidinger [1].[1] In this approach arrays of Taylor coefficients of various functions are referred to as AD variables or *pyramids* since, as will be seen, they have a hyper-pyramidal structure. Generally the first entry in the array will be the value of the function about some expansion point, and the remaining entries will be the higher-order Taylor coefficients about the expansion point and truncated beyond some specified order. Such truncated Taylor expansions are also commonly called *jets*. Recall Section 7.5.

In our application elements in these arrays will be addressed and manipulated with the aid of scalar indices associated with look-up tables generated at run time. We have also replaced the original APL implementation of Neidinger with a code written in the language of *Mathematica* (Version 6, or 7) [2,3]. Where necessary, for those unfamiliar with the details of *Mathematica*, we will explain the consequences of various *Mathematica* commands. Recall that we wish to integrate equations of the form

$$\dot{z}_a = f_a(\boldsymbol{z}, t), \quad a = 1, m \tag{S.1.1}$$

and their associated complete variational equations. The inputs to the code are the right sides (RS) of (1.1). Other input parameters are the number of variables m, the desired order of the Taylor map p, and the initial conditions $(z_a^d)^i$ for the design solution.

Various AD tools for describing and manipulating pyramids are outlined in Section S.2. There we show how pyramid operations are encoded in the case of polynomial RS, as needed, for example, for the Duffing equation. For brevity, we omit the cases of rational, fractional power, and transcendental RS. These cases can also be handled using various methods based on functional identities and known Taylor coefficients, or the differential equations that such

[1]Some authors refer to AD as *truncated power series algebra* (TPSA) since AD algorithms arise from manipulating multivariable truncated power series. Other authors refer to AD as *Differential Algebra* (DA). There is a substantial literature on this subject. See the Web site http://www.autodiff.org/.

functions obey along with certain recursion relations [1]. In Section S.3, based on the work of Section S.2, we in effect obtain and integrate numerically the complete variational equations (10.12.36) in pyramid form, i.e. valid for any map order and any number of variables. Section S.4 treats the specific case of the Duffing equation. A final Section S.5 describes in more detail the relation between integrating equations for pyramids and the complete variational equations.

S.2 AD Tools

This section describes how arithmetic expressions representing $f_a(\boldsymbol{z},t)$, the right sides of (1.1) where $\boldsymbol{z}$ denotes the dependent variables, are replaced with expressions for arrays (pyramids) of Taylor coefficients. These pyramids in turn constitute the input to our code. Such an ad-hoc replacement, according to the problem at hand, as opposed to operator overloading where the kind of operation depends on the type of its argument, is also the approach taken in [1,4,5].

Let u, v, w be general arithmetic expressions, i.e. scalar-valued functions of $\boldsymbol{z}$. They contain various arithmetic operations such as addition/subtraction ($\pm$), multiplication ($*$), and raising to a power ($\wedge$). (They may also entail the computation of various transcendental functions such as the sine function, etc. However, as stated earlier, for simplicity we will omit these cases.) The arguments of these operations may be a constant, a single variable or multiple variables z_a, or even some other expression. The idea of AD is to redefine the arithmetic operations in such a way (see Definition 1), that all functions u, v, w can be consistently replaced with the arrays of coefficients of their Taylor expansions. For example, by redefining the usual product of numbers ($*$) and introducing the pyramid operation `PROD`, $u * v$ is replaced with `PROD[U,V]`.

We use upper typewriter font for pyramids (`U,V,...`) and for operations on pyramids (`PROD, POW, ...`). Everywhere, equalities written in typewriter fonts have equivalent *Mathematica* expressions. That is, they have associated realizations in *Mathematica* and directly correspond to various operations and commands in *Mathematica*. In effect, our code operates entirely on pyramids. However, as we will see, any pyramid expression contains, as its first entry, its usual arithmetic counterpart.

We begin with a description of our method of monomial labeling. In brief, we list all monomials in a polynomial in some sequence, and label them by where they occur in the list. Next follow Definition 1 and the recipes for encoding operations on pyramids. Subsequently, by using Definition 2, which simply states the rule by which an arithmetic expression is replaced with its pyramid counterpart, we show how a general expression can be encoded by using only the pyramids of a constant and those of the various variables involved.

S.2.1 Labeling Scheme

A monomial $G_{\boldsymbol{j}}(\boldsymbol{z})$ in m variables is of the form

$$G_{\boldsymbol{j}}(\boldsymbol{z}) = (z_1)^{j_1}(z_2)^{j_2}\cdots(z_m)^{j_m}. \tag{S.2.1}$$

Here we have introduced an exponent vector $\boldsymbol{j}$ by the rule

$$\boldsymbol{j} = (j_1, j_2, \cdots j_m). \tag{S.2.2}$$

Evidently $\boldsymbol{j}$ is an m-tuple of non-negative integers. The degree of $G_{\boldsymbol{j}}(\boldsymbol{z})$, denoted by $|\boldsymbol{j}|$, is given by the sum of exponents,

$$|\boldsymbol{j}| = j_1 + j_2 + \cdots + j_m. \tag{S.2.3}$$

The set of all exponents for monomials in m variables with degree less than or equal to p will be denoted by Γ_m^p,

$$\Gamma_m^p = \{\boldsymbol{j} \mid |\boldsymbol{j}| \leq p\}. \tag{S.2.4}$$

According to Section 32.1, this set has $L(m,p)$ entries with $L(m,p)$ given by a binomial coefficient,

$$L(m,p) = S_0(m,p) = \binom{p+m}{p}. \tag{S.2.5}$$

With this notation, a Taylor series expansion (about the origin) of a scalar-valued function u of m variables $\boldsymbol{z} = (z_1, z_2, \ldots z_m)$, truncated beyond terms of degree p, can be written in the form

$$u(\boldsymbol{z}) = \sum_{\boldsymbol{j} \in \Gamma_m^p} \mathtt{U}(\boldsymbol{j})\ G_{\boldsymbol{j}}(\boldsymbol{z}). \tag{S.2.6}$$

Assuming that m and p are fixed input variables, we will often simply write Γ and L. Here, for now, $\mathtt{U}$ simply denotes an array of numerical coefficients. When employed in code that has symbolic manipulation capabilities, each $\mathtt{U}(\boldsymbol{j})$ may also be a symbolic quantity.

To proceed, what is needed is some way of listing monomials systematically. With such a list, as described in Subsections 32.3.3 and 32.3.4, we may assign a label r to each monomial based on where it appears in the list. We will use a variant of *modified glex sequencing*, the only change being that we will begin the list with the monomial of degree 0. For example, Table 2.1 shows a list of monomials in three variables. As one goes down the list, first the monomial of degree $D = 0$ appears, then the monomials of degree $D = 1$, etc. Within each group of monomials of fixed degree the individual monomials appear in descending lex order. Note that Table 2.1 is similar to Table 32.2.4 except that it begins with the monomial of degree 0. Other possible listings include ascending true glex order in which monomials appear in ascending lex order within each group of degree D, and lex order for the whole monomial list as in [1].

Table S.2.1: A labeling scheme for monomials in three variables.

r	j_1	j_2	j_3	D
1	0	0	0	0
2	1	0	0	1
3	0	1	0	1
4	0	0	1	1
5	2	0	0	2
6	1	1	0	2
7	1	0	1	2
8	0	2	0	2
9	0	1	1	2
10	0	0	2	2
11	3	0	0	3
12	2	1	0	3
13	2	0	1	3
14	1	2	0	3
15	1	1	1	3
16	1	0	2	3
17	0	3	0	3
18	0	2	1	3
19	0	1	2	3
20	0	0	3	3
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
28	1	2	1	4
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

With the aid of the scalar index r the relation (2.6) can be rewritten in the form

$$u(\boldsymbol{z}) = \sum_{r=1}^{L(m,p)} \mathtt{U}(r) G_r(\boldsymbol{z}), \tag{S.2.7}$$

because (by construction and with fixed m) for each positive integer r there is a unique exponent $\boldsymbol{j}(r)$, and for each $\boldsymbol{j}$ there is a unique r. Here `U` may be viewed as a vector with entries `U(r)`, and $G_r(\boldsymbol{z})$ denotes $G_{\boldsymbol{j}(r)}(\boldsymbol{z})$.

Consider, in an m-dimensional space, the points defined by the heads of the vectors $\boldsymbol{j} \in \Gamma_m^p$. See (2.4). Figure 2.1 displays them in the case $m = 3$ and $p = 4$. Evidently they form a grid that lies on the surface and interior of what can be viewed as an m-dimensional *pyramid* in m-dimensional space. At each grid point there is an associated coefficient `U(r)`.

Because of its association with this pyramidal structure, we will refer to the entire set of coefficients in (2.6) or (2.7) as the *pyramid* U of $u(\boldsymbol{z})$.

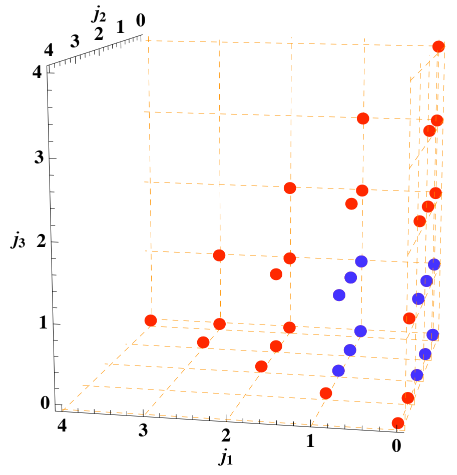

Figure S.2.1: A grid of points representing the set Γ_3^4. For future reference a subset of Γ_3^4, called a *box*, is shown in blue.

S.2.2 Implementation of Labeling Scheme

We have seen that use of modified glex sequencing, for any specified number of variables m, provides a labeling rule such that for each positive integer r there is a unique exponent $\boldsymbol{j}(r)$, and for each $\boldsymbol{j}$ there is a unique r. That is, there is a invertible function $r(\boldsymbol{j})$ that provides a 1-to-1 correspondence between the positive integers and the exponent vectors $\boldsymbol{j}$. To proceed further, it would be useful to have this function and its inverse in more explicit form.

From the work of Subsection 32.2.6, we already know a formula for $r(\boldsymbol{j})$ based on the Giorgilli formula (32.2.15),

$$r(\boldsymbol{j}) = r(j_1, \cdots j_m) = 1 + i(j_1, \cdots j_m). \tag{S.2.8}$$

Below is simple *Mathematica* code that implements this formula (which we call *Gfor*) in the case of three variables, and evaluates it for selected exponents $\boldsymbol{j}$. Observe that these evaluations agree with results in Table 2.1.

```
Gfor[j1_, j2_, j3_] := (
s1 = j3; s2 = 1 + j3 + j2; s3 = 2 + j3 + j2 + j1;
t1 = Binomial[s1, 1]; t2 = Binomial[s2, 2]; t3 = Binomial[s3, 3];
r = 1 + t1 + t2 + t3; r
)
Gfor[0, 0, 0]
Gfor[1, 0, 0]
Gfor[2, 0, 1]
Gfor[1, 2, 1]
1
2
13
28
```
(S.2.9)

For the inverse relation we have found it convenient to introduce a rectangular matrix associated with the set Γ_m^p. By abuse of notation, it will also be called Γ. It has $L(m,p)$ rows and m columns with entries

$$\Gamma_{r,a} = j_a(r). \tag{S.2.10}$$

For example, looking a Table 2.1, we see (when $m = 3$) that $\Gamma_{1,1} = 0$ and $\Gamma_{17,2} = 3$. Indeed, if the first and last columns of Table 2.1 are removed, what remains (when $m = 3$) is the matrix $\Gamma_{r,a}$. In the language of Subsection 32.2.9, Γ is a *look up table* that, given r, produces the associated $\boldsymbol{j}$. In our *Mathematica* implementation Γ is the matrix `GAMMA` with elements `GAMMA[[r, a]]`.

The matrix `GAMMA` is constructed using the *Mathematica* code illustrated below,

```
Needs["Combinatorica`"];
m = 3; p = 4;
GAMMA = Compositions[0, m];
Do[GAMMA = Join[GAMMA, Reverse[Compositions[d, m]]], {d, 1, p, 1}];
L = Length[GAMMA]
r = 17; a = 2;
GAMMA[[r]]
GAMMA[[r, a]]
35
{0, 3, 0}
3
```
(S.2.11)

It employs the *Mathematica* commands `Compositions`, `Reverse`, and `Join`.

We will now describe the ingredients of this code and illustrate the function of each:

- The command `Needs["Combinatorica`"];` loads a combinatorial package.
- The command `Compositions[i, m]` produces, as a list of arrays (a rectangular array), all *compositions* (under addition) of the integer i into m integer parts. Furthermore, the compositions appear in *ascending* lex order. For example, the command `Compositions[0, 3]` produces the single row

$$0 \;\; 0 \;\; 0 \tag{S.2.12}$$

As a second example, the command `Compositions[1, 3]` produces the rectangular array

$$\begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{array} \tag{S.2.13}$$

As a third example, the command `Compositions[2, 3]` produces the rectangular array

$$\begin{array}{ccc} 0 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 2 & 0 & 0 \end{array} \tag{S.2.14}$$

- The command `Reverse` acts on the list of arrays, and reverses the order of the list while leaving the arrays intact. For example, the nested sequence of commands `Reverse[Compositions[1, 3]]` produces the rectangular array

$$\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \tag{S.2.15}$$

As a second example, the nested sequence of commands `Reverse[Compositions[2, 3]]` produces the rectangular array

$$\begin{array}{ccc} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{array} \tag{S.2.16}$$

Now the compositions appear in *descending* lex order.

- Look, for example, at Table 2.1. We see that the exponents j_a for the $r = 1$ entry are those appearing in (2.12). Next, exponents for the $r = 2$ through $r = 4$ entries are those appearing in (2.15). Following them, the exponents for the $r = 5$ through $r = 10$ entries, are those appearing in (2.16), etc. Evidently, to produce the exponent list of Table 2.1, what we must do is successively *join* various lists. That is what the *Mathematica* command `Join` accomplishes.

We are now ready to describe how `GAMMA` is constructed:

- The second line in (2.11) sets the values of m and p. They are assigned the values $m = 3$ and $p = 4$ for this example, which will construct `GAMMA` for the case of Table 2.1. The third line in (2.11) initially sets `GAMMA` to a row of m zeroes. The fourth line is a `Do` loop that successively redefines `GAMMA` by generating and joining to it successive descending lex order compositions. The net result is the exponent list of Table 2.1.

- The quantity $L = L(m, p)$ is obtained by applying the *Mathematica* command `Length` to the the rectangular array `GAMMA`.

- The last 6 lines of (2.11) illustrate that L is computed properly and that the command `GAMMA[[r, a]]` accesses the array `GAMMA` in the desired fashion. Specifically, in this example, we find from (2.5) that $L(3, 4) = 35$ in agreement with the *Mathematica* output for L. Moreover, `GAMMA[[17]]` produces the exponent array $\{0, 3, 0\}$, in agreement with the $r = 17$ entry in Table 2.1, and `GAMMA[[17, 2]]` produces $\Gamma_{17,2} = 3$, as expected.

S.2.3 Pyramid Operations: General Procedure

Here we *derive* the pyramid operations in terms of $\boldsymbol{j}$-vectors by using the ordering previously described, and provide scripts to *encode* them in the r-representation (2.7).

Definition 1. Suppose that $w(\boldsymbol{z})$ arises from carrying out various *arithmetic operations* on $u(\boldsymbol{z})$ and $v(\boldsymbol{z})$, and the associated pyramids `U` and `V` are known. The corresponding pyramid operation on `U` and `V` is so defined that it yields the pyramid `W` of $w(\boldsymbol{z})$.

Here we assume that u, v, w are polynomials such as (2.6).

S.2.4 Pyramid Operations: Scalar Multiplication and Addition

We begin with the operations of scalar multiplication and addition, which are easy to define and implement. If

$$w(\boldsymbol{z}) = c\, u(\boldsymbol{z}), \tag{S.2.17}$$

then

$$\mathtt{W}(r) = c\, \mathtt{U}(r), \tag{S.2.18}$$

and we write

$$\mathtt{W} = c\, \mathtt{U}. \tag{S.2.19}$$

If

$$w(\boldsymbol{z}) = u(\boldsymbol{z}) + v(\boldsymbol{z}), \tag{S.2.20}$$

then

$$\mathtt{W}(r) = \mathtt{U}(r) + \mathtt{V}(r), \tag{S.2.21}$$

and we write

$$\mathtt{W} = \mathtt{U} + \mathtt{V}. \tag{S.2.22}$$

In both cases all operations are performed coordinate-wise (as for vectors).

Implementation of scalar multiplication and vector addition is easy in *Mathematica* because, as the example below illustrates, it has built in vector routines. There we define two vectors, multiply them by scalars, and add the resulting vectors.

```
Unprotect[V];
U = {1, 2, 3};
V = {4, 5, 6};
W = .1U + .2V
{.9, 1.2, 1.5}                                  (S.2.23)
```

Since V is a "protected" symbol in the *Mathematica* language, and, for purposes of illustration, we wish to use it as an ordinary vector variable, it must first be unprotected as in line 1 above. The last line shows that the *Mathematica* output is indeed the desired result.

S.2.5 Pyramid Operations: Background for Polynomial Multiplication

The operation of polynomial multiplication is more involved. Now we have the relation

$$w(\boldsymbol{z}) = u(\boldsymbol{z}) * v(\boldsymbol{z}), \tag{S.2.24}$$

and we want to encode

$$\mathtt{W} = \mathtt{PROD}[\mathtt{U}, \mathtt{V}]. \tag{S.2.25}$$

Shown below is *Mathematica* code that implements this operation,

```
PROD[U_, V_] := Table[U[[B[[k]]]] · V[[Brev[[k]]]], {k, 1, L, 1}];      (S.2.26)
```

Our next task is to describe and explain the ingredients in (2.26).

Let us write $u(\boldsymbol{z})$ in the form (2.6), but with a change of dummy indices, so that it has the representation

$$u(\boldsymbol{z}) = \sum_{\boldsymbol{i} \in \Gamma_m^p} \mathtt{U}(\boldsymbol{i})\, G_{\boldsymbol{i}}(\boldsymbol{z}). \tag{S.2.27}$$

Similarly, write $v(\boldsymbol{z})$ in the form

$$v(\boldsymbol{z}) = \sum_{\boldsymbol{j} \in \Gamma_m^p} \mathtt{V}(\boldsymbol{j})\, G_{\boldsymbol{j}}(\boldsymbol{z}). \tag{S.2.28}$$

Then, according to Leibniz, there is the result

$$u(\boldsymbol{z}) * v(\boldsymbol{z}) = \sum_{\boldsymbol{i} \in \Gamma_m^p} \sum_{\boldsymbol{j} \in \Gamma_m^p} \mathtt{U}(\boldsymbol{i})\mathtt{V}(\boldsymbol{j}) G_{\boldsymbol{i}}(\boldsymbol{z}) * G_{\boldsymbol{j}}(\boldsymbol{z}). \tag{S.2.29}$$

From (2.1) we observe that

$$\begin{aligned} G_{\boldsymbol{i}}(\boldsymbol{z}) * G_{\boldsymbol{j}}(\boldsymbol{z}) &= (z_1)^{i_1}(z_2)^{i_2}\cdots(z_m)^{i_m} * (z_1)^{j_1}(z_2)^{j_2}\cdots(z_m)^{j_m} \\ &= (z_1)^{i_1+j_1}(z_2)^{i_2+j_2}\cdots(z_m)^{i_m+j_m} = G_{\boldsymbol{i}+\boldsymbol{j}}(\boldsymbol{z}). \end{aligned} \tag{S.2.30}$$

Therefore, we may also write

$$u(\boldsymbol{z}) * v(\boldsymbol{z}) = \sum_{\boldsymbol{i}\,\in\,\Gamma_m^p}\ \sum_{\boldsymbol{j}\,\in\,\Gamma_m^p} \mathtt{U}(\boldsymbol{i})\mathtt{V}(\boldsymbol{j})G_{\boldsymbol{i}+\boldsymbol{j}}(\boldsymbol{z}). \tag{S.2.31}$$

Now we see that there are two complications. First, there may be terms on the right side of (2.31) whose degree is higher than p and therefore need not be computed. Second, there are generally many terms on the right side of (2.31) that contribute to a given monomial term in $w(\boldsymbol{z}) = u(\boldsymbol{z}) * v(\boldsymbol{z})$. Suppose we write

$$w(\boldsymbol{z}) = \sum_{\boldsymbol{k}} \mathtt{W}(\boldsymbol{k})\ G_{\boldsymbol{k}}(\boldsymbol{z}). \tag{S.2.32}$$

Upon comparing (2.31) and (2.32) we conclude that

$$\mathtt{W}(\boldsymbol{k}) = \sum_{\boldsymbol{i}+\boldsymbol{j}=\boldsymbol{k}} \mathtt{U}(\boldsymbol{i})\mathtt{V}(\boldsymbol{j}) = \sum_{\boldsymbol{j}\le\boldsymbol{k}} \mathtt{U}(\boldsymbol{k}-\boldsymbol{j})\mathtt{V}(\boldsymbol{j}). \tag{S.2.33}$$

Here, by $\boldsymbol{j} \le \boldsymbol{k}$, we mean that the sum ranges over all $\boldsymbol{j}$ such that $j_a \le k_a$ for all $a \in [1, m]$. That is,

$$\boldsymbol{j} \le \boldsymbol{k} \ \Leftrightarrow\ j_a \le k_a \text{ for all } a \in [1, m]. \tag{S.2.34}$$

Evidently, to implement the relation (2.33) in terms of r labels, we need to describe the exponent relation $\boldsymbol{j} \le \boldsymbol{k}$ in terms of r labels. Suppose $\boldsymbol{k}$ is some exponent vector with label $r(\boldsymbol{k})$ as, for example, in Table 2.1. Introduce the notation

$$k = r(\boldsymbol{k}). \tag{S.2.35}$$

This notation may be somewhat confusing because k is not the norm of the vector $\boldsymbol{k}$, but rather the label associated with $\boldsymbol{k}$. However, this notation is very convenient. Now, given a label k, we can find $\boldsymbol{k}$. Indeed, from (2.10), we have the result

$$k_a = \Gamma_{k,a}. \tag{S.2.36}$$

Having found $\boldsymbol{k}$, we define a set of exponents B_k by the rule

$$B_k = \{\boldsymbol{j} | \boldsymbol{j} \le \boldsymbol{k}\}. \tag{S.2.37}$$

This set of exponents is called the k^{th} *box*. Note that the heads of the vectors $\boldsymbol{j}$ that satisfy (2.37) for some fixed vector $\boldsymbol{k}$ do indeed lie within some hyper-rectangular volume (box). For example (when $m = 3$), suppose $k = 28$. Then we see from Table 2.1 that $\boldsymbol{k} = (1, 2, 1)$. Table 2.2 lists, in modified glex order, all the vectors in B_{28}, i.e. all vectors $\boldsymbol{j}$ such that

Table S.2.2: The vectors in $B_{28} = \{\boldsymbol{j} | \boldsymbol{j} \leq (1,2,1)\}$.

r	j_1	j_2	j_3	D
1	0	0	0	0
2	1	0	0	1
3	0	1	0	1
4	0	0	1	1
6	1	1	0	2
7	1	0	1	2
8	0	2	0	2
9	0	1	1	2
14	1	2	0	3
15	1	1	1	3
18	0	2	1	3
28	1	2	1	4

$\boldsymbol{j} \leq (1,2,1)$. These are the vectors whose heads are shown in blue in Figure 2.1. Finally, with this notation, we can rewrite (2.33) in the form

$$\mathtt{W}(\boldsymbol{k}) = \sum_{\boldsymbol{j} \in B_k} \mathtt{U}(\boldsymbol{k} - \boldsymbol{j}) \mathtt{V}(\boldsymbol{j}). \tag{S.2.38}$$

What can be said about the vectors $(\boldsymbol{k} - \boldsymbol{j})$ as $\boldsymbol{j}$ ranges over B_ℓ? Table 2.3 lists, for example, the vectors $\boldsymbol{j} \in B_{28}$ and the associated vectors $\boldsymbol{i}$ with $\boldsymbol{i} = (\boldsymbol{k} - \boldsymbol{j})$. Also listed are the labels $r(\boldsymbol{j})$ and $r(\boldsymbol{i})$. Compare columns 2,3,4, which specify the $\boldsymbol{j} \in B_{28}$, with columns 5,6,7, which specify the associated $\boldsymbol{i}$ vectors. We see that every vector that appears in the $\boldsymbol{j}$ list also occurs somewhere in the $\boldsymbol{i}$ list, and vice versa. This to be expected because the operation of multiplication is commutative: we can also write (2.33) in the form

$$\mathtt{W}(\boldsymbol{k}) = \sum_{\boldsymbol{j} \in B_k} \mathtt{U}(\boldsymbol{j}) \mathtt{V}(\boldsymbol{k} - \boldsymbol{j}). \tag{S.2.39}$$

We also observe the more remarkable feature that the two lists are *reverses* of each other: running down the $\boldsymbol{j}$ list gives the same vectors as running up the $\boldsymbol{i}$ list, and vice versa. This feature is a consequence of our ordering procedure.

As indicated earlier, what we really want is a version of (2.33) that involves labels instead of exponent vectors. Looking at Table 2.3, we see that this is easily done. We may equally well think of B_k as containing a collection of labels $r(\boldsymbol{j})$, and we may introduce a *reversed* array $Brev_k$ of *complementary* labels $r^c(\boldsymbol{j})$ where

$$r^c(\boldsymbol{j}) = r(\boldsymbol{i}). \tag{S.2.40}$$

That is, for example, B_{28} would consist of the first column of Table 2.3 and $Brev_{28}$ would consist of the last column of Table 2.3. Finally, we have already introduced k as being the label associated with $\boldsymbol{k}$. We these understandings in mind, we may rewrite (2.33) in the label form

$$\mathtt{W}(k) = \sum_{r \in B_k} \mathtt{U}(r^c)\mathtt{V}(r) = \sum_{r \in B_k} \mathtt{U}(r)\mathtt{V}(r^c). \tag{S.2.41}$$

This is the rule $\mathtt{W} = \mathtt{PROD}[\mathtt{U}, \mathtt{V}]$ for multiplying pyramids. In the language of Section 32.7, B_k and $Brev_k$, when taken together, provide a *look back table* that, given k, look back to find all monomial pairs with labels r, r^c which produce, when multiplied, the monomial with label k.

Table S.2.3: The vectors $\boldsymbol{j}$ and $\boldsymbol{i} = (\boldsymbol{k} - \boldsymbol{j})$ for $\boldsymbol{j} \in B_{28}$ and $k_a = \Gamma_{28,a}$.

$r(\boldsymbol{j})$	j_1	j_2	j_3	i_1	i_2	i_3	$r(\boldsymbol{i})$
1	0	0	0	1	2	1	28
2	1	0	0	0	2	1	18
3	0	1	0	1	1	1	15
4	0	0	1	1	2	0	14
6	1	1	0	0	1	1	9
7	1	0	1	0	2	0	8
8	0	2	0	1	0	1	7
9	0	1	1	1	1	0	6
14	1	2	0	0	0	1	4
15	1	1	1	0	1	0	3
18	0	2	1	1	0	0	2
28	1	2	1	0	0	0	1

S.2.6 Pyramid Operations: Implementation of Multiplication

The code shown below in (2.42) illustrates how B_k and $Brev_k$ are constructed using *Mathematica.*

```
JSK[list_, K_] :=
Position[Apply[And, Thread[#1<=#2&[#,K]]]& /@ list, True]//Flatten;
B = Table[JSK[GAMMA, GAMMA[[k]]], {k, 1, L}];
Brev = Reverse /@ B;                                        (S.2.42)
```

As before, some explanation is required. The main tasks are to implement the $\boldsymbol{j} \leq \boldsymbol{k}$ operation (2.34) and then to employ this implementation. We will begin by implementing the $\boldsymbol{j} \leq \boldsymbol{k}$ operation. Several steps are required, and each of them is described briefly below:

- When *Mathematica* is presented with a statement of the form $j <= k$, with j and k being *integers*, it replies with the answer True or the answer False. (Here $j <= k$

denotes $j \leq k$.) Two sample *Mathematica* runs are shown below:

```
3 <= 4
True
```
(S.2.43)

```
5 <= 4
False
```
(S.2.44)

- A *Mathematica* function can be constructed that does the same thing. It takes the form

```
#1 <= #2 & [j, k]
```
(S.2.45)

Here the symbols `#1` and `#2` set up two *slots* and the symbol `&` means the operation to its left is to be regarded as a function and is to be applied to the arguments to its right by inserting the arguments into the slots. Below is a *Mathematica* run illustrating this feature.

```
j = 3; k = 4;
#1 <= #2 & [j, k]
True
```
(S.2.46)

Observe that the output of this run agrees with that of (2.43).

- The same operation can be performed on pairs of *arrays* (rather than pairs of numbers) in such a way that corresponding entries from each array are compared, with the output then being an array of True and False answers. This is done using the *Mathematica* command `Thread`. Below is a *Mathematica* run illustrating this feature.

```
j = {1, 2, 3}; k = {4, 5, 1};
Thread[#1 <= #2 & [j, k]]
{True, True, False}
```
(S.2.47)

Note that the first two answers in the output array are True because the statements $1 \leq 4$ and $2 \leq 5$ are true. The last answer in the output array is False because the statement $3 \leq 1$ is false.

- Suppose, now, that we are given two arrays $\boldsymbol{j}$ and $\boldsymbol{k}$ and we want to determine if $\boldsymbol{j} \leq \boldsymbol{k}$ in the sense of (2.34). This can be done by *applying* the logical `And` operation (using the *Mathematica* command `Apply`) to the True/False output array described above. Below is a *Mathematica* run illustrating this feature.

```
j = {1, 2, 3}; k = {4, 5, 1};
Apply[And, Thread[#1 <= #2 & [j, k]]]
False
```
(S.2.48)

Note that the output answer is False because at least one of the entries in the output array in (2.47) is False. The output answer would be True if, and only if, all entries in the output array in (2.47) were True.

- Now that the $\boldsymbol{j} \leq \boldsymbol{k}$ operation has been defined for two exponent arrays, we would like to construct a related operator/function, to be called `JSK`. (Here the letter `S` stands for *smaller than or equal to.*) It will depend on the exponent array $\boldsymbol{k}$, and its task will be to search a list of exponent arrays to find those $\boldsymbol{j}$ within it that satisfy $\boldsymbol{j} \leq \boldsymbol{k}$. The first step in this direction is to slightly modify the function appearing in (2.48). Below is a *Mathematica* run that specifies this modified function and illustrates that it has the same effect.

```
j = {1, 2, 3}; k = {4, 5, 1};
Apply[And, Thread[#1 <= #2 & [#, k]]] & [j]
False                                              (S.2.49)
```

  Comparison of the functions in (2.48) and (2.49) reveals that what has been done is to replace the argument j in (2.48) by a slot `#`, then follow the function by the character `&`, and finally add the symbols [j]. What this modification does is to redefine the function in such a way that it acts on what follows the second `&`.

- The next step is to extend the function appearing in (2.49) so that it acts on a list of exponent arrays. To do this, we replace the symbols [j] by the symbols `/@ list`. The symbols `/@` indicate that what stands to their left is to act on what stands to their right, and what stands to their right is a list of exponent arrays. The result of this action will be a list of True/False results with one result for each exponent array in the list. Below is a *Mathematica* run that illustrates how the further modified function acts on lists.

```
k = {4, 5, 1};
ja = {3, 4, 1}; jb = {1, 2, 3}; jc = {1, 2, 1};
list = {ja, jb, jc};
Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list
{True, False, True}                                (S.2.50)
```

  Observe that the output answer list is {True, False, True} because $\{3, 4, 1\} \leq \{4, 5, 1\}$ is True, $\{1, 2, 3\} \leq \{4, 5, 1\}$ is False, and $\{1, 2, 1\} \leq \{4, 5, 1\}$ is True.

- What we would really like to know is where the True items are in the list, because that will tell us where the $\boldsymbol{j}$ that satisfy $\boldsymbol{j} \leq \boldsymbol{k}$ reside. This can be accomplished by use of the *Mathematica* command `Position` in conjunction with the result True. Below is a *Mathematica* run that illustrates how this works.

```
k = {4, 5, 1};
ja = {3, 4, 1}; jb = {1, 2, 3}; jc = {1, 2, 1};
list = {ja, jb, jc};
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]
{{1}, {3}}                                         (S.2.51)
```

Note that the output is an array of positions in the list for which $\boldsymbol{j} \leq \boldsymbol{k}$. There is, however, still one defect. Namely, the output array is an array of single-element subarrays, and we would like it to be simply an array of location numbers. This defect can be remedied by appending the *Mathematica* command `Flatten`, preceded by `//`, to the instruction string in (2.51). The *Mathematica* run below illustrates this modification.

```
k = {4, 5, 1};
ja = {3, 4, 1}; jb = {1, 2, 3}; jc = {1, 2, 1};
list = {ja, jb, jc};
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]//Flatten
{1, 3}                                                        (S.2.52)
```

Now the output is a simple array containing the positions in the list for which $\boldsymbol{j} \leq \boldsymbol{k}$.

- The last step is to employ the ingredients in (2.52) to define the operator `JSK[list, k]`. The *Mathematica* run below illustrates how this can be done.

```
k = {4, 5, 1};
ja = {3, 4, 1}; jb = {1, 2, 3}; jc = {1, 2, 1};
list = {ja, jb, jc};
JSK[list_, k_] :=
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]//Flatten;
JSK[list, k]
{1, 3}                                                        (S.2.53)
```

Lines 4 and 5 above define the operator `JSK[list, k]`, line 6 invokes it, and line 7 displays its output, which agrees with the output of (2.52).

- With the operator `JSK[list, k]` in hand, we are prepared to construct tables B and $Brev$ that will contain the B_k and the $Brev_k$. The *Mathematica* run below illustrates how this can be done.

```
B = Table[JSK[GAMMA, GAMMA[[k]]], {k, 1, L, 1}];
Brev = Reverse /@ B;
B[[8]]
Brev[[8]]
B[[28]]
Brev[[28]]
{1, 3, 8}
{8, 3, 1}
{1, 2, 3, 4, 6, 7, 8, 9, 14, 15, 18, 28}
{28, 18, 15, 14, 9, 8, 7, 6, 4, 3, 2, 1}                      (S.2.54)
```

The first line employs the *Mathematica* command `Table` in combination with an implied Do loop to produce a two-dimensional array `B`. Values of k in the range $[1, L]$ are selected sequentially. For each k value the associated exponent array $\boldsymbol{k}(k) = \texttt{GAMMA[[k]]}$ is obtained. The operator `JSK` then searches the full `GAMMA` array to find the list of r values associated with the $\boldsymbol{j} \leq \boldsymbol{k}$. All these r values are listed in a row. Thus, the array `B` consists of list of L rows, of varying width. The rows are labeled by $k \in [1, L]$, and in each row are the r values associated with the $\boldsymbol{j} \leq \boldsymbol{k}$. In the second line the *Mathematica* command `Reverse` is applied to `B` to produce a second array called `Brev`. Its rows are the reverse of those in `B`. For example, as the *Mathematica* run illustrates, `B[[8]]`, which is the 8th row of `B`, contains the list $\{1, 3, 8\}$, and `Brev[[8]]` contains the list $\{8, 3, 1\}$. Inspection of the $r = 8$ monomial in Table 2.1, that with exponents $\{0, 2, 0\}$, shows that it has the monomials with exponents {0,0,0}, {0,1,0}, and {0,2,0} as factors. And further inspection of Table 2.1 shows that the exponents of these factors have the r values $\{1, 3, 8\}$. Similarly `B[[28]]`, which is the 28th row of B, contains the same entries that appear in the first column of Table 2.3. And `Brev[[28]]`, which is the 28th row of *Brev*, contains the same entries that appear in the last column of Table 2.3.

Finally, we need to explain how the arrays B and *Brev* can be employed to carry out polynomial multiplication. This can be done using the *Mathematica* dot product command:

- The exhibit below shows a simple *Mathematica* run that illustrates the use of the dot product command.

```
Unprotect[V];
U = {.1, .2, .3, .4, .5, .6, .7, .8};
V = {1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8};
U.V
u = {1, 3, 5};
v = {6, 4, 2};
U[[u]]
V[[v]]
U[[u]].V[[v]]
5.64
{.1, .3, .5}
{1.6, 1.4, 1.2}
1.18
```
(S.2.55)

As before, `V` must be unprotected. See line 1. The rest of the first part this run (lines 2 through 4) defines two vectors `U` and `V` and then computes their dot product. Note that if we multiply the entries in `U` and `V` pairwise and add, we get the result

$$.1 \times 1.1 + .2 \times 1.2 + \cdots + .8 \times 1.8 = 5.64,$$

which agrees with the *Mathematica* result for `U·V`. See line 10.

The second part of this *Mathematica* run, lines 5 through 9, illustrates a powerful feature of the *Mathematica* language. Suppose, as illustrated, we define two arrays `u` and `v` of integers, and use these arrays as *arguments* for the vectors by writing `U[[u]]` and `V[[v]]`. Then *Mathematica* uses the integers in the two arrays `u` and `v` as labels to select the corresponding entries in `U` and `V`, and from these entries it makes new corresponding vectors. In this example, the 1^{st}, 3^{rd}, and 5^{th} entries in `U` are .1, .3, and .5. And the 6^{th}, 4^{th}, and 2^{nd} entries in `V` are 1.6, 1.4, and 1.2. Consequently, we find that

$$\mathtt{U}[[\mathtt{u}]] = \{.1, .3, .5\},$$

$$\mathtt{V}[[\mathtt{v}]] = \{1.6, 1.4, 1.2\},$$

in agreement with lines 11 and 12 of the *Mathematica* results. Correspondingly, we expect that `U[[u]]·V[[v]]` will have the value

$$\mathtt{U}[[\mathtt{u}]]\cdot\mathtt{V}[[\mathtt{v}]] = .1 \times 1.6 + .3 \times 1.4 + .5 \times 1.2 = 1.18,$$

in agreement with the last line of the *Mathematica* output.

- Now suppose, as an example, that we set $k = 8$ and use `B[[k]]` and `Brev[[k]]` in place of the arrays `u` and `v`. The *Mathematica* fragment below shows what happens when this is done.

```
k = 8;
B[[k]]
Brev[[k]]
U[[B[[k]]]]
V[[Brev[[k]]]]
U[[B[[k]]]] · V[[Brev[[k]]]]
{1, 3, 8}
{8, 3, 1}
{.1, .3, .8}
{1.8, 1.3, 1.1}
1.45                                   (S.2.56)
```

From (2.54) we see that `B[[8]]` $= \{1, 3, 8\}$ and `Brev[[8]]` $= \{8, 3, 1\}$ in agreement with lines 7 and 8 of the *Mathematica* output above. Also, the 1^{st}, 3^{rd}, and 8^{th} entries in `U` are .1, .3, and .8. And the 8^{th}, 3^{rd}, and 1^{st} entries in `V` are 1.8, 1.3, and 1.1. Therefore we expect the results

$$\mathtt{U}[[\mathtt{B}[[\mathtt{k}]]]] = \{.1, .3, .8\},$$

$$\mathtt{V}[[\mathtt{Brev}[[\mathtt{k}]]]] = \{1.8, 1.3, 1.1\},$$

$$\mathtt{U}[[\mathtt{B}[[\mathtt{k}]]]]\cdot\mathtt{V}[[\mathtt{Brev}[[\mathtt{k}]]]] = .1 \times 1.8 + .3 \times 1.3 + .8 \times 1.1 = 1.45,$$

in agreement with the last three lines of (2.56).

- Finally, suppose we carry out the operation `U[[B[[k]]]] · V[[Brev[[k]]]]` for all $k \in [1, L]$ and put the results together in a Table with entries labeled by k. According to (2.41), the result will be the pyramid for the product of the two polynomials whose individual pyramids are `U` and `V`. The *Mathematica* fragment (2.26), which is displayed again below, shows how this can be done to define a *product* function, called `PROD`, that acts on general pyramids `U` and `V`, using the command Table with an implied Do loop over k.

```
PROD[U_, V_] := Table[U[[B[[k]]]] · V[[Brev[[k]]]], {k, 1, L, 1}];
```

Let us verify that this whole multiplication procedure works for a simple example. For the sake of brevity, we will consider the case of $m = 2$ variables and work through terms of degree $p = 3$. In this case pyramids have the modest length $L(2, 3) = 10$. Table 2.4 provides a labeling scheme for monomials in two variables using our standard modified glex sequencing.

Table S.2.4: A labeling scheme for monomials in two variables.

r	j_1	j_2
1	0	0
2	1	0
3	0	1
4	2	0
5	1	1
6	0	2
7	3	0
8	2	1
9	1	2
10	0	3
·	·	·
·	·	·

Suppose, for example, that u and v are the functions

$$u(\boldsymbol{z}) = 1 + 2z_1 + 3z_2 + 4z_1z_2 \tag{S.2.57}$$

and

$$v(\boldsymbol{z}) = 5 + 6z_1 + 7z_2^2. \tag{S.2.58}$$

From Table 2.4 we find that the corresponding pyramids `U` and `V` are

$$\mathtt{U} = \{1, 2, 3, 0, 4, 0, 0, 0, 0, 0\} \tag{S.2.59}$$

and

$$\mathtt{V} = \{5, 6, 0, 7, 0, 0, 0, 0, 0, 0\}. \tag{S.2.60}$$

Polynomial multiplication gives the result

$$\begin{aligned} w(\boldsymbol{z}) &= u(\boldsymbol{z}) * v(\boldsymbol{z}) \\ &= 5 + 16z_1 + 15z_2 + 12z_1^2 + 38z_1z_2 + 7z_2^2 + 24z_1^2z_2 + 14z_1z_2^2 + 21z_2^3 + 28z_1z_2^3. \end{aligned} \tag{S.2.61}$$

Correspondingly, through terms of degree 3, the pyramid W = PROD[U, V] is given by

$$\mathtt{W} = \{5, 16, 15, 12, 38, 7, 0, 24, 14, 21\}. \tag{S.2.62}$$

Below is an execution of a *Mathematica* program illustrating the use of the product function for the polynomials u and v given by (2.57) and (2.58).

```
Clear["Global‘ * "];
Needs["Combinatorica‘"];
m = 2; p = 3;
GAMMA = Compositions[0, m];
Do[GAMMA = Join[GAMMA, Reverse[Compositions[d, m]]], {d, 1, p, 1}];
L = Length[GAMMA]
JSK[list_, k_] :=
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]//Flatten;
B = Table[JSK[GAMMA, GAMMA[[r]]], {r, 1, L, 1}];
Brev = Reverse/@ B;
PROD[U_, V_] := Table[U[[B[[k]]]].V[[Brev[[k]]]], {k, 1, L, 1}];
U = {1, 2, 3, 0, 4, 0, 0, 0, 0, 0};
V = {5, 6, 0, 7, 0, 0, 0, 0, 0, 0};
10
PROD[U, V]
{5, 16, 15, 12, 38, 7, 0, 24, 14, 21}                                   (S.2.63)
```

The first 11 lines of the code set up the necessary arrays and define the product function in pyramid form. The next two lines specify the pyramids U and V given in (2.59) and (2.60). The third line from the bottom, which results from the command in line 6, illustrates that indeed $L(2, 3) = 10$. The final two lines show that use of the product function when applied to the pyramids U and V does indeed product the pyramid W given by (2.62).

S.2.7 Pyramid Operations: Implementation of Powers

With operation of multiplication in hand, it is easy to implement the operation of raising a pyramid to a power. The code shown below in (2.64) demonstrates how this can be done.

```
POWER[U_, 0] := C1;
POWER[U_, 1] := U;
POWER[U_, 2] := PROD[U, U];
POWER[U_, 3] := PROD[U, POWER[U, 2]];
...                                          (S.2.64)
```

Here `C1` is the pyramid for the Taylor series having *one* as its *constant* term and all other terms zero,

$$\texttt{C1} = \{1, 0, 0, 0, \cdots\}. \tag{S.2.65}$$

It can be set up by the *Mathematica* code

```
C1 = Table[KroneckerDelta[k, 1], {k, 1, L, 1}];      (S.2.66)
```

which employs the Table command, the Kronecker delta function, and an implied Do loop over k. This code should be executed before executing (2.64), but after the value of L has been established.

S.2.8 Replacement Rule and Automatic Differentiation

Definition 2. The transformation $A(\boldsymbol{z}) \rightsquigarrow \texttt{A}$ means replacement of every real variable z_a in the *arithmetic expression* $A(\boldsymbol{z})$ with an associated pyramid, and of every operation on real variables in $A(\boldsymbol{z})$ with the associated operation on pyramids.

Automatic differentiation is based on the following corollary: if $A(\boldsymbol{z}) \rightsquigarrow \texttt{A}$, then `A` is the pyramid of $A(\boldsymbol{z})$.

For simplicity, we will begin our discussion of the replacement rule with examples involving only a single variable z. In this case monomial labeling, the relation between labels and exponents, is given directly by the simple rules

$$r(j) = 1 + j \text{ and } j(r) = r - 1. \tag{S.2.67}$$

See Table 2.5.

As a first example, consider the expression

$$A = 2 + 3(z * z). \tag{S.2.68}$$

We have agreed to consider the case $m = 1$. Suppose we also set $p = 2$, in which case $L = 3$. In ascending glex order, see Table 2.5, the pyramid for A is then

$$2 + 3z^2 \rightsquigarrow \texttt{A} = (2, 0, 3). \tag{S.2.69}$$

Now imagine that A was not such a simple polynomial, but some complicated expression. Then the pyramid `A` could be generated by computing derivatives of A at $z = 0$ and dividing

Table S.2.5: A labeling scheme for monomials in one variable.

r	j
1	0
2	1
3	2
4	3
.	.
.	.

them by the appropriate factorials. Automatic differentiation offers another way to find `A`. Assume that all operations in the arithmetic expression A have been encoded according to *Definition 1*. For our example, these are + and `PROD`. Let `C1` and `Z` be the pyramids associated with 1 and z,

$$1 \rightsquigarrow \mathtt{C1} = (1, 0, 0), \tag{S.2.70}$$

$$z \rightsquigarrow \mathtt{Z} = (\mathtt{0}, \mathtt{1}, \mathtt{0}). \tag{S.2.71}$$

The quantity $2+3z^2$ results from performing various arithmetic operations on 1 and z. *Definition 1* says that the pyramid of $2+3z^2$ is identical to the pyramid obtained by performing the same operations on the pyramids `C1` and `Z`. That is, suppose we replace 1 and z with their associated pyramids `C1` and `Z`, and also replace $*$ with `PROD`. Then, upon evaluating `PROD`, multiplying by the appropriate scalar coefficients, and summing, the result will be the same pyramid `A`,

$$2\ \mathtt{C1} + 3\ \mathtt{PROD}[\mathtt{Z}, \mathtt{Z}] = \mathtt{A}. \tag{S.2.72}$$

In this way, by knowing only the basic pyramids `C1` and `Z` (prepared beforehand), one can compute the pyramid of an arbitrary $A(z)$. Finally, in contrast to numerical differentiation, all numerical operations involved are accurate to machine precision. *Mathematica* code that implements (2.72) will be presented shortly in (2.73).

Frequently, if $A(z)$ is some complicated expression, the replacement rule will result in a long chain of nested pyramid operations. At every step in the chain the present pyramid, the pyramid resulting from the previous step, will be combined with some other pyramid to produce a new pyramid. Each such operation has two arguments (the present pyramid and some other pyramid), and *Definition 1* applies to each step in the chain. Upon evaluating all pyramid operations, the final result will be the pyramid of $A(z)$.

By using the replacement operation the above procedure can be represented as:

$$1 \rightsquigarrow \mathtt{C1}, \quad z \rightsquigarrow \mathtt{Z}, \quad A \rightsquigarrow \mathtt{A}.$$

The following general recipe then applies: In order to derive the pyramid associated with some arithmetic expression, apply the $\rightsquigarrow$ rule to all its variables, or parts, and replace all operations with operations on pyramids. Here "apply the $\rightsquigarrow$ rule" to something means replace that something with the associated pyramid. And the term "parts" means subexpressions. *Definition 1* guarantees that the result will be the same pyramid `A` no matter how we split the arithmetic expression A into subexpressions. It is only necessary to recognize, in case of using subexpressions, that one pyramid expression should be viewed as a function of another.

For illustration, suppose we regard the A given by (2.68) to be the composition of two functions, $F(z) = 2 + 3z$ and $G(z) = z^2$, so that $A(z) = F(G(z))$. Instead of associating a constant and a single variable with their respective pyramids, let us now associate whole subexpressions. In addition, let us label the pyramid expressions on the right of $\rightsquigarrow$ with with some names, `F` and `G`:

$$2 + 3z \rightsquigarrow 2\ \mathtt{C1} + 3\ \mathtt{Z} = \mathtt{F[Z]}$$

$$z^2 \rightsquigarrow \mathtt{PROD[Z,Z]} = \mathtt{G[Z]}$$

$$A(z) \rightsquigarrow \mathtt{F[G[Z]]} = \mathtt{A}.$$

We have indicated the explicit dependence on `Z`. It is important to note that `F[Z]` is a pyramid *expression* prior to executing any the pyramid operations, i.e it is not yet a pyramid, but is simply the result of formal replacements that follow the association rule.

Mathematica code for the simple example (2.72) is shown below,

```
C1 = {1, 0, 0};
Z = {0, 1, 0};
2 C1 + 3 PROD[Z, Z]
{2, 0, 3}                                    (S.2.73)
```

Note that the result (2.73) agrees with (2.69). This example does not use any nested expressions. We will now illustrate how the same results can be obtained using nested expressions.

We begin by displaying a simple *Mathematica* program/execution, that employs ordinary variables, and uses *Mathematica*'s intrinsic abilities to handle nested expressions. The program/execution is

```
f[z_] := 2 + 3z;
g[z_] := z^2;
f[g[z]]
2 + 3z^2                                     (S.2.74)
```

With *Mathematica* the underscore in `z_` indicates that `z` is a dummy variable name, and the symbols := indicate that `f` is defined with a delayed assignment. That is what is done in line one above. The same is done in line two for `g`. Line three requests evaluation of the nested function $f(g(z))$, and the result of this evaluation is displayed in line four. Note that the result agrees with (2.68).

With this background, we are ready to examine a program with analogous nested pyramid operations. The same comments apply regarding the use of underscores and delayed assignments. The program is

```
C1 = {1, 0, 0};
Z = {0, 1, 0};
F[Z_] := 2 C1 + 3 Z;
G[Z_] := PROD[Z, Z];
F[G[Z]]
{2, 0, 3}
```
(S.2.75)

Note that line (2.75) agrees with line (2.73), and is consistent with line (2.69).

S.2.9 Taylor Rule

We close this section with an important consequence of the replacement rule and nested operations, which we call the *Taylor* rule. We begin by considering functions of a single variable. Suppose the function $G(x)$ has the special form

$$G(x) = z^d + x \tag{S.2.76}$$

where z^d is some constant. Let F be some other function. Consider the composite (nested) function A defined by

$$A(x) = F(G(x)) = F(z^d + x). \tag{S.2.77}$$

Then, assuming the necessary analyticity and by the chain rule, A evidently has a Taylor expansion in x about the origin of the form

$$\begin{aligned} A &= A(0) + A'(0)x + (1/2)A''(0)x^2 + \cdots \\ &= F(z^d) + F'(z^d)x + (1/2)F''(z^d)x^2 + \cdots . \end{aligned} \tag{S.2.78}$$

We conclude that if we know the Taylor expansion of A about the origin, then we also know the Taylor expansion of F about z^d, and vice versa. Suppose, for example, that

$$F(z) = 1 + 2z + 3z^2 \tag{S.2.79}$$

and

$$z^d = 4. \tag{S.2.80}$$

Then there is the result

$$A(x) = F(G(x)) = F(z^d + x) = 1 + 2(4 + x) + 3(4 + x)^2 = 57 + 26x + 3x^2. \tag{S.2.81}$$

We now show that this same result can be obtained using pyramids. The *Mathematica* fragment below illustrates how this can be done.

```
C1 = {1, 0, 0};
X = {0, 1, 0};
zd = 4;
F[Z_] := 1 C1 + 2 Z + 3 PROD[Z, Z];
G[X_] := zd C1 + X;
F[G[X]]
{57, 26, 3}
```
(S.2.82)

Note that (2.82) agrees with (2.81). See also Table 2.5.

Let us also illustrate the Taylor rule in the two-variable case. Let $F(z_1, z_2)$ be some function of two variables. Introduce the functions $G(x_1)$ and $H(x_1)$ having the special forms

$$G(x_1) = z_1^d + x_1, \tag{S.2.83}$$

$$H(x_2) = z_2^d + x_2, \tag{S.2.84}$$

where z_1^d and z_2^d are some constants. Consider the function A defined by

$$A(x_1, x_2) = F(G(x_1), H(x_2)) = F(z_1^d + x_1, z_2^d + x_2). \tag{S.2.85}$$

Then, again assuming the necessary analyticity and by the chain rule, A evidently has a Taylor expansion in x_1 and x_2 about the origin $(0, 0)$ of the form

$$\begin{aligned} A &= A(0,0) + [\partial_1 A(0,0)]x_1 + [\partial_2 A(0,0)]x_2 \\ &\quad +(1/2)[(\partial_1)^2 A(0,0)]x_1^2 + [\partial_1\partial_2 A(0,0)]x_1x_2 + (1/2)[(\partial_2)^2 A(0,0)]x_2^2 + \cdots \\ &= F(z_1^d, z_2^d) + [\partial_1 F(z_1^d, z_2^d)]x_1 + [\partial_2 F(z_1^d, z_2^d)]x_2 \\ &\quad +(1/2)[(\partial_1)^2 F(z_1^d, z_2^d)]x_1^2 + [\partial_1\partial_2 AF(z_1^d, z_2^d)]x_1x_2 + (1/2)[(\partial_2)^2 A(F(z_1^d, z_2^d))]x_2^2 + \cdots \end{aligned} \tag{S.2.86}$$

where

$$\partial_1 = \partial/\partial x_1, \;\; \partial_2 = \partial/\partial x_2 \tag{S.2.87}$$

when acting on A, and

$$\partial_1 = \partial/\partial z_1, \;\; \partial_2 = \partial/\partial z_2 \tag{S.2.88}$$

when acting on F. We conclude that if we know the Taylor expansion of A about the origin $(0, 0)$, then we also know the Taylor expansion of F about (z_1^d, z_2^d), and vice versa.

As a concrete example, suppose that

$$F(z_1, z_2) = 1 + 2z_1 + 3z_2 + 4z_1^2 + 5z_1z_2 + 6z_2^2 \tag{S.2.89}$$

and

$$z_1^d = 7, \;\; z_2^d = 8. \tag{S.2.90}$$

Then, hand calculation shows that $F(G(x_1), H(x_2))$ takes the form

$$\begin{aligned} F(z_1^d + x_1, z_2^d + x_2) &= F(G(x_1), H(x_2)) \\ &= 899 + 98x_1 + 4x_1^2 + 134x_2 + 5x_1x_2 + 6x_2^2. \end{aligned} \tag{S.2.91}$$

Below is a *Mathematica* execution that finds the same result,

```
F[z1_, z2_] := 1 + 2 z1 + 3 z2 + 4 z1^2 + 5 z1 z2 + 6 z2^2
G[x1_] := zd1 + x1;
H[x2_] := zd2 + x2;
zd1 = 7;
zd2 = 8;
A = F[G[x1], H[x2]]
Expand[A]
1 + 2 (7 + x1) + 4 (7 + x1)^2 + 3 (8 + x2) + 5 (7 + x1) (8 + x2) + 6 (8 + x2)^2
899 + 98 x1 + 4 x1^2 + 134 x2 + 5 x1 x2 + 6 x2^2
```
(S.2.92)

The calculation above dealt with the case of a function of two ordinary variables. We now illustrate, for the same example, that there is an analogous result for pyramids. Following the replacement rule, we should make the substitutions

$$z_1^d + x_1 \rightsquigarrow \mathtt{zd1\ C1} + \mathtt{X1}, \tag{S.2.93}$$

$$z_2^d + x_2 \rightsquigarrow \mathtt{zd2\ C1} + \mathtt{X2}, \tag{S.2.94}$$

$$\begin{aligned}&1 + 2\ z_1 + 3\ z_2 + 4\ z_1^2 + 5\ z_1\ z_2 + 6\ z_2^2 \rightsquigarrow\\ &\mathtt{C1} + 2\ \mathtt{Z1} + 3\ \mathtt{Z2} + 4\ \mathtt{PROD[Z1, Z1]} + 5\ \mathtt{PROD[Z1, Z2]} + 6\ \mathtt{PROD[Z2, Z2]}.\end{aligned} \tag{S.2.95}$$

The *Mathematica* fragment below, executed for the case $m = 2$ and $p = 2$, in which case $L = 6$, illustrates how the analogous result is obtained using pyramids,

```
C1 = {1, 0, 0, 0, 0, 0};
X1 = {0, 1, 0, 0, 0, 0};
X2 = {0, 0, 1, 0, 0, 0};
F[Z1_, Z2_] := C1 + 2 Z1 + 3 Z2 + 4 PROD[Z1, Z1] + 5 PROD[Z1, Z2]
+6 PROD[Z2, Z2];
G[X1_] := z01 C1 + X1;
H[X2_] := z02 C1 + X2;
zd1 = 7;
zd2 = 8;
F[G[X1], H[X2]]
{899, 98, 134, 4, 5, 6}
```
(S.2.96)

Note that, when use is made of Table 2.4, the last line of (2.96) agrees with (2.91) and the last line of (2.92).

S.3 Numerical Integration and Replacement Rule

S.3.1 Numerical Integration

Consider the set of differential equations (1.1). As described in Chapter 2, a standard procedure for their numerical integration from an initial time $t^i = t^0$ to some final time t^f is to divide the time axis into a large number of steps N, each of small duration h, thereby introducing successive times t^n defined by the relation

$$t^n = t^0 + nh \;\text{ with }\; n = 0, 1, \cdots, N. \tag{S.3.1}$$

By construction, there will also be the relation

$$Nh = t^f - t^i. \tag{S.3.2}$$

The goal is to compute the vectors $\boldsymbol{z}^n$, where

$$\boldsymbol{z}^n = \boldsymbol{z}(t^n), \tag{S.3.3}$$

starting from the vector $\boldsymbol{z}^0$. The vector $\boldsymbol{z}^0$ is assumed given as a set of definite numbers, i.e. the initial conditions at t^0.

If we assume for the solution piece-wise analyticity in t, or at least sufficient differentiability in t (which will be the case if the f_a are piece-wise analytic or at least have sufficient differentiability in t), we may convert the set of differential equations (1.1) into a set of recursion relations for the $\boldsymbol{z}^n$ in such a way that the $\boldsymbol{z}^n$ obtained by solving the recursion relations differ from the true $\boldsymbol{z}^n$ by only small truncation errors of order h^m. (Here m is *not* the number of variables, but rather some fixed integer describing the accuracy of the integration method.) One such procedure, a fourth-order *Runge Kutta* (RK4) method, is the set of marching/recursion rules

$$\boldsymbol{z}^{n+1} = \boldsymbol{z}^n + \frac{1}{6}(\boldsymbol{a} + 2\boldsymbol{b} + 2\boldsymbol{c} + \boldsymbol{d}) \tag{S.3.4}$$

where, at each step,

$$\begin{aligned} \boldsymbol{a} &= h\boldsymbol{f}(\boldsymbol{z}^n, t^n), \\ \boldsymbol{b} &= h\boldsymbol{f}(\boldsymbol{z}^n + \frac{1}{2}\boldsymbol{a}, t^n + \frac{1}{2}h), \\ \boldsymbol{c} &= h\boldsymbol{f}(\boldsymbol{z}^n + \frac{1}{2}\boldsymbol{b}, t^n + \frac{1}{2}h), \\ \boldsymbol{d} &= h\boldsymbol{f}(\boldsymbol{z}^n + \boldsymbol{c}, t^n + h). \end{aligned} \tag{S.3.5}$$

Thanks to the genius of Runge and Kutta, the relations (3.4) and (3.5) have been constructed in such a way that the method is locally (at each step) correct through order h^4, and makes local truncation errors of order h^5. Recall Section 2.3.2

In the case of a single variable, and therefore a single differential equation, the relations (3.4) and (3.5) may be encoded in the *Mathematica* form shown below. Here `Zvar` is the dependent variable, `t` is the time, `Zt` is a temporary variable, `tt` is a temporary time, and `ns` is the number of integration steps. The program employs a Do loop over `i` so that the operations (3.4) and (3.5) are carried out `ns` times.

```
RK4 := (
   t0 = t;
Do[
   Aa = h F[Zvar, t];
   Zt = Zvar + (1/2)Aa;
   tt = t + h/2;
   Bb = h F[Zt, tt];
   Zt = Zvar + (1/2)Bb;
   Cc = h F[Zt, tt];
   Zt = Zvar + Cc;
   tt = t + h;
   Dd = h F[Zt, tt];
   Zvar = Zvar + (1/6)(Aa + 2 Bb + 2 Cc + Dd);
   t = t0 + i h;,
   {i, 1, ns, 1}
   ]
 )
```

(S.3.6)

S.3.2 Replacement Rule, Single Equation/Variable Case

We now make what, for our purposes, is a fundamental observation: The operations that occur in the Runge Kutta recursion rules (3.4) and (3.5) and realized in the code above can be extended to pyramids by application of the replacement rule. In particular, the dependent variable $\boldsymbol{z}$ can be replaced by a pyramid, and the various operations involved in the recursion rules can be replaced by pyramid operations. Indeed if we look at the code above, apart from the evaluation of `F`, we see that the quantities `Zvar`, `Zt`, `Aa`, `Bb`, `Cc`, and `Dd` can be viewed, if we wish, as pyramids since the only operations involved are scalar multiplication and addition. The only requirement for a pyramidal interpretation of the `RK4` *Mathematica* code is that the right side of the differential equation, `F[*, *]`, be defined for pyramids. Finally, we remark that the features that make it possible to interpret the `RK4` *Mathematica* code either in terms of ordinary variables or pyramidal variables will hold for *Mathematica* realizations of many other familiar numerical integration methods including other forms of Runge Kutta, predictor-corrector methods, and extrapolation methods.

To make these ideas concrete, and to understand their implications, let us begin with a simple example. Suppose, in the single variable case, that the right side of the differential equation has the simple form

$$f(z, t) = -2tz^2. \tag{S.3.7}$$

The differential equation with this right side can be integrated analytically to yield the solution

$$z(t) = z^0/[1 + z^0(t - t^0)^2]. \tag{S.3.8}$$

In particular, for the case $t^0 = 0$, $z^0 = 1$, and $t = 1$, there is the result

$$z(1) = z^0/[1 + z^0] = 1/2. \tag{S.3.9}$$

Let us also integrate the differential equation with the right side (3.7) numerically. Shown below is the result of running the associated *Mathematica* Runge Kutta code for this case.

```
Clear["Global` * "];
F[Z_, t_] := −2 t Z^2;
h = .1;
ns = 10;
t = 0;
Zvar = 1.;
RK4;
t
Zvar
1.
0.500001
```

(S.3.10)

Note that the last line of (3.10) agrees with (3.9) save for a "1" in the last entry. As expected, and as experimentation shows, this small difference, due to accumulated truncation error, becomes even smaller if `h` is decreased (and correspondingly, `ns` is increased).

Suppose we expand the solution (3.9) about the design initial condition $z^{d0} = 1$ by replacing z^0 by $z^{d0} + x$ and expanding the result in a Taylor series in x about the point x=0. Below is a *Mathematica* run that performs this task.

```
zd0 = 1;
Series[(zd0 + x)/(1 + zd0 + x), {x, 0, 5}]
```

$$\frac{1}{2} + \frac{x}{4} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{x^4}{32} + \frac{x^5}{64} + O[x]^6$$

(S.3.11)

We will now see that the same Taylor series can be obtained by the operation of numerical integration applied to pyramids. The *Mathematica* code below shows, for our example differential equation, the application of numerical integration to pyramids.

```
Clear["Global` * "];
Needs["Combinatorica`"];
m = 1; p = 5;
GAMMA = Compositions[0, m];
Do[GAMMA = Join[GAMMA, Reverse[Compositions[d, m]]], {d, 1, p, 1}];
L = Length[GAMMA];
JSK[list_, k_] :=
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]//Flatten;
B = Table[JSK[GAMMA, GAMMA[[r]]], {r, 1, L, 1}];
Brev = Reverse/@ B;
PROD[U_, V_] := Table[U[[B[[k]]]].V[[Brev[[k]]]], {k, 1, L, 1}];
F[Z_, t_] := −2 t PROD[Z, Z];
h = .01;
ns = 100;
t = 0;
zd0 = 1;
C1 = {1, 0, 0, 0, 0, 0};
X = {0, 1, 0, 0, 0, 0};
Zvar = zd0 C1 + X;
RK4;
t
Zvar
1.
{0.5, 0.25, −0.125, 0.0625, −0.03125, 0.015625}                (S.3.12)
```

The first 11 lines of the code set up what should be by now the familiar procedure for labeling and multiplying pyramids. In particular, $m = 1$ because we are dealing with a single variable, and $p = 5$ since we wish to work through fifth order. The line

$$\texttt{F[Z\_, t\_] := −2 t PROD[Z, Z]} \tag{S.3.13}$$

defines `F[∗, ∗]` for the case of pyramids, and is the result of applying the replacement rule to the right side of f as given by (3.7),

$$-2\,t\,z^2 \rightsquigarrow -2\;\texttt{t PROD[Z, Z]}. \tag{S.3.14}$$

Lines 13 through 15 play the same role as lines 3 through 5 in (3.10) except that, in order to improve numerical accuracy, the step size `h` has been decreased and correspondingly the number of steps `ns` has been increased. Lines 16 through 19 now initialize `Zvar` as a pyramid with a constant part `zd0` and first-order monomial part with coefficient 1,

$$\texttt{Zvar = zd0 C1 + X}. \tag{S.3.15}$$

These lines are the pyramid equivalent of line 6 in (3.10). Finally lines 20 through 22 are the same as lines 7 through 9 in (3.10). In particular, the line `RK4` in (3.10) and the line `RK4` in (3.12) refer to exactly the *same* code, namely that in (3.6).

Let us now compare the outputs of (3.10) and (3.12). Comparing the penultimate lines in each we see that the final time $t = 1$ is the same in each case. Comparing the last lines shows that the output `Zvar` for (3.12) is a pyramid whose first entry agrees with the last line of (3.10). Finally, all the entries in the pyramid output agree with the Taylor coefficients in the expansion (3.11). We see, in the case of numerical integration (of a single differential equation), that replacing the dependent variable by a pyramid, with the initial value of the pyramid given by (3.15), produces a Taylor expansion of the final condition in terms of the initial condition.

What accounts for this near miraculous result? It's the Taylor rule described described in Subsection 2.9. We have already learned that to expand some function $F(z)$ about some point z^d we must evaluate $F(z^d + x)$. See (2.77). We know that the final $Zvar$, call it $Zvar^{\text{fin}}$, is an analytic function of the initial $Zvar$, call it $Zvar^{\text{in}}$, so that we may write

$$Zvar^{\text{fin}} = Zvar^{\text{fin}}(Zvar^{\text{in}}) = g(Zvar^{\text{in}}) \tag{S.3.16}$$

where g is the function that results from following the trajectory from $t = t^{\text{in}}$ to $t = t^{\text{fin}}$. Therefore, by the Taylor rule, to expand $Zvar^{\text{fin}}$ about $Zvar^{\text{in}} = z^{d0}$, we must evaluate $Zvar^{\text{fin}}(z^{d0} + x)$. That, with the aid of pyramids, is what the code (3.12) accomplishes.

S.3.3 Multi Equation/Variable Case

Because of *Mathematica's* built-in provisions for handling arrays, the work of the previous section can easily be extended to the case of several differential equations. Consider, as an example, the two-variable case for which $\boldsymbol{f}$ has the form

$$\begin{aligned} f_1(\boldsymbol{z}, t) &= -z_1^2, \\ f_2(\boldsymbol{z}, t) &= +2z_1 z_2. \end{aligned} \tag{S.3.17}$$

The differential equations associated with this $\boldsymbol{f}$ can be solved in closed form to yield, with the understanding that $t^0 = 0$, the solution

$$\begin{aligned} z_1(t) &= z_1^0/(1 + tz_1^0), \\ z_2(t) &= z_2^0(1 + tz_1^0)^2. \end{aligned} \tag{S.3.18}$$

For the final time $t = 1$ we find the result

$$\begin{aligned} z_1(1) &= z_1^0/(1 + z_1^0), \\ z_2(1) &= z_2^0(1 + z_1^0)^2. \end{aligned} \tag{S.3.19}$$

Let us expand the solution (3.19) about the design initial conditions

$$\begin{aligned} z_1^{d0} &= 1, \\ z_2^{d0} &= 2, \end{aligned} \tag{S.3.20}$$

by writing

$$\begin{aligned} z_1^0 &= z_1^{d0} + x_1 = 1 + x_1, \\ z_2^0 &= z_2^{d0} + x_2 = 2 + x_2. \end{aligned} \tag{S.3.21}$$

Doing so gives the results

$$\begin{aligned} z_1(1) &= (1 + x_1)/(2 + x_1) = (2 + x_1 - 1)/(2 + x_1) = 1 - 1/(2 + x_1) = \\ &= 1 - (1/2)(1 + x_1/2)^{-1} = 1 - (1/2)[1 - x_1/2 + (x_1/2)^2 - (x_1/2)^3 + \cdots \\ &= (1/2) + (1/4)x_1 - (1/8)x_1^2 + (1/16)x_1^3 + \cdots , \end{aligned} \tag{S.3.22}$$

$$\begin{aligned} z_2(1) &= (2 + x_2)(2 + x_1)^2 \\ &= 8 + 8x_1 + 4x_2 + 2x_1^2 + 4x_1x_2 + x_1^2x_2. \end{aligned} \tag{S.3.23}$$

We will now explore how this same result can be obtained using the replacement rule applied to the operation of numerical integration. As before, we will label individual monomials by an integer r. Recall that Table 2.5 shows our standard modified glex sequencing applied to the case of two variables.

The *Mathematica* code below shows, for our two-variable example differential equation, the application of numerical integration to pyramids. Before describing the code in some detail, we take note of the bottom two lines. When interpreted with the aid of Table 2.4, we see that the penultimate line of (3.24) agrees with (3.22), and the last line of (3.24) nearly agrees with (3.23). The only discrepancy is that for the monomial with label $r = 7$ in the last line of (3.24). In the *Mathematica* output it has the value -1.16563×10^{-7} while, according to (3.23), the true value should be zero. This small discrepancy arises from the truncation error inherent in the RK4 algorithm, and becomes smaller as the step size `h` is decreased (and `ns` is correspondingly increased), or if some more accurate integration algorithm is used. We conclude that, with the use of pyramids, it is also possible in the two-variable case to obtain Taylor expansions of the final conditions in terms of the initial conditions. Indeed, what is involved is again the Taylor rule applied, in this instance, to the case of two variables.

```
Clear["Global` * "];
Needs["Combinatorica`"];
m = 2; p = 3;
GAMMA = Compositions[0, m];
Do[GAMMA = Join[GAMMA, Reverse[Compositions[d, m]]], {d, 1, p, 1}];
L = Length[GAMMA];
JSK[list_, k_] :=
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]//Flatten;
B = Table[JSK[GAMMA, GAMMA[[r]]], {r, 1, L, 1}];
Brev = Reverse/@ B;
PROD[U_, V_] := Table[U[[B[[k]]]].V[[Brev[[k]]]], {k, 1, L, 1}];
F[Z_, t_] := {-PROD[Z[[1]], Z[[1]]], 2. PROD[Z[[1]], Z[[2]]]};
h = .01;
ns = 100;
t = 0;
zd0 = {1., 2.};
C1 = Table[KroneckerDelta[k, 1], {k, 1, L, 1}];
X[1] = Table[KroneckerDelta[k, 2], {k, 1, L, 1}];
X[2] = Table[KroneckerDelta[k, 3], {k, 1, L, 1}];
Zvar = {zd0[[1]] C1 + X[1], zd0[[2]] C1 + X[2]};
RK4;
t
Zvar
1.
{{0.5, 0.25, 0., -0.125, 0., 0., 0.0625, 0., 0., 0, },
  {8., 8., 4., 2., 4., 0., -1.16563 × 10^-7, 1., 0., 0.}}
```

(S.3.24)

Let us compare the structures of the routines for the single variable case and multi (two) variable case as illustrated in (3.12) and (3.24). The first difference occurs at line 3 where the number of variables m and the maximum degree p are specified. In (3.24) m is set to 2 because we wish to treat the case of two variables, and p is set to 3 simply to limit the lengths of the output arrays. The next difference occurs in line 12 where the right side `F` of the differential equation is specified. The major feature of the definition of `F` in (3.24) is that it is specified as two pyramids because the right side of the definition has the structure $\{*, *\}$ where each item $*$ is an instruction for computing a pyramid. In particular, the two pyramids are those for the two components of $\boldsymbol{f}$ as given by (3.17) and use of the replacement rule,

$$-z_1^2 \rightsquigarrow -\texttt{PROD[Z[[1]], Z[[1]]]}, \tag{S.3.25}$$

$$2z_1z_2 \rightsquigarrow 2.\,\texttt{PROD[Z[[1]],Z[[2]]]}. \tag{S.3.26}$$

The next differences occur in lines 16 through 20 of (3.24). In line 16, since specification of the initial conditions now requires two numbers, see (3.20), `zd0` is specified as a two-component array. In lines 17 and 18 of (3.12) the pyramids `C1` and `X` are set up explicitly for the case $p = 5$. By contrast, in lines 17 through 19 of (3.24), the pyramids `C1`, `X[1]`, and `X[2]` are set up for general p with the aid of the Table command and the Kronecker delta function. Recall (2.66) and observe from Tables 2.1, 2.4, and 2.5 that, no matter what the values of m and p, the constant monomial has the label $r = 1$ and the monomial x_1 has the label $r = 2$. Moreover, as long as $m \geq 2$ and no matter what the value of p, the x_2 monomial has the label $r = 3$. Finally, compare line 19 in (3.12) with line 20 in (3.24), both of which define the initial `Zvar`. We see that the difference is that in (3.12) `Zvar` is defined as a single pyramid while in (3.24) it is defined as a pair of pyramids of the form $\{*, *\}$. Most remarkably, all other corresponding lines in (3.12) and (3.24) are the same. In particular, the *same* RK4 code, namely that given by (3.6), is used in the scalar case (3.10), the single pyramid case (3.12), and the two-pyramid case (3.24). This multi-use is possible because of the convenient way in which *Mathematica* handles arrays.

We conclude that the pattern for the multivariable case is now clear. Only the following items need to be specified in an m dependent way:

- The value of m.
- The entries in `F` with entries entered as an array $\{*, *, \cdots\}$ of m pyramids.
- The design initial condition array `zd0`.
- The pyramids for `C1` and `X[1]` through `X[m]`.
- The entries for the initial `Zvar` specified as an array

  $\{\texttt{zd0[[1]] C1} + \texttt{X[1]}, \texttt{zd0[[2]] C1} + \texttt{X[2]}, \cdots, \texttt{zd0[[m]] C1} + \texttt{X[m]}\}$ of m pyramids.

S.4 Duffing Equation Application

Let us now apply the methods just developed to the case of the Duffing equation with parameter dependence as described by the relations (10.12.133) through (10.12.138). *Mathematica* code for this purpose is shown below. By looking at the final lines that result from executing this code, we see that the final output is an array of the form $\{\{*\}, \{*\}, \{*\}\}$. That is, the final output is an array of three pyramids. This is what we expect, because now we are dealing with three variables. See line 3 of the code, which sets $m = 3$. Also, for convenience of viewing, results are calculated and displayed only through third order as a consequence of setting $p = 3$.

```
Clear["Global` * "];
Needs["Combinatorica`"];
m = 3; p = 3;
GAMMA = Compositions[0, m];
Do[GAMMA = Join[GAMMA, Reverse[Compositions[d, m]]], {d, 1, p, 1}];
L = Length[GAMMA];
JSK[list_, k_] :=
Position[Apply[And, Thread[#1 <= #2 & [#, k]]] & /@ list, True]//Flatten;
B = Table[JSK[GAMMA, GAMMA[[r]]], {r, 1, L, 1}];
Brev = Reverse/@ B;
PROD[U_, V_] := Table[U[[B[[k]]]].V[[Brev[[k]]]], {k, 1, L, 1}];
POWER[U_, 2] := PROD[U, U];
POWER[U_, 3] := PROD[U, POWER[U, 2]];
C0 = Table[0, {k, 1, L, 1}];
F[Z_, t_] := {Z[[2]],
−2. beta PROD[Z[[3]], Z[[2]]] − PROD[POWER[Z[[3]], 2], Z[[1]]]−
POWER[Z[[1]], 3] − eps Sin[t] POWER[Z[[3]], 3],
C0};
ns = 100;
t = 0;
h = (2Pi)/ns;
beta = .1; eps = 1.5;
zd0 = {.3, .4, .5};
C1 = Table[KroneckerDelta[k, 1], {k, 1, L, 1}];
X[1] = Table[KroneckerDelta[k, 2], {k, 1, L, 1}];
X[2] = Table[KroneckerDelta[k, 3], {k, 1, L, 1}];
X[3] = Table[KroneckerDelta[k, 4], {k, 1, L, 1}];
Zvar = {zd0[[1]] C1 + X[1], zd0[[2]] C1 + X[2], zd0[[3]] C1 + X[3]};
RK4;
t
Zvar
```

$$\begin{aligned}
&2\pi\\
&\{\{-0.0493158, 0.973942, -0.110494, 5.51271, 3.54684, 3.46678,\\
&\quad 11.2762, 2.36463, 1.0985, 23.3332, -1.03541, -3.23761, -12.8064,\\
&\quad 4.03421, -23.4342, -17.8967, 1.96148, 5.07403, -36.9009, 25.1379\},\\
&\ \{0.439713, 1.05904, 0.427613, 3.3177, 0.0872459, 0.635397, -3.02822,\\
&\quad 1.77416, -4.10115, 3.16981, -2.43002, -5.33643, -7.77038, -6.08476,\\
&\quad -0.541465, -21.1672, -1.4091, -9.54326, 14.6334, -39.2312\},\\
&\ \{0.5, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}\}
\end{aligned} \tag{S.4.1}$$

The first unusual fragments in the code are lines 12 and 13, which define functions that implement the calculation of second and third powers of pyramids. Recall Subsection 2.7. The first new fragment is line 14, which defines the pyramid `C0` with the aid of the Table command and an implied Do loop. As a result of executing this code, `C0` is an array of L zeroes. The next three lines, lines 15 through 18, define `F`, which specifies the right sides of equations (10.12.133) through (10.12.135). See (10.12.136) through (10.12.138). The right side of `F` is of the form $\{*, *, *\}$, an array of three pyramids. By looking at (10.12.136) and recalling the replacement rule, we see that the first pyramid should be `Z[[2]]`,

$$z_2 \rightsquigarrow \texttt{Z[[2]]}. \tag{S.4.2}$$

The second pyramid on the right side of `F` is more complicated. It arises by applying the replacement rule to the right side of (10.12.137) to obtain the associated pyramid,

$$\begin{aligned}
&-2\beta z_3 z_2 - z_3^2 z_1 - z_1^3 - \epsilon z_3^3 \sin t \rightsquigarrow\\
&-2.\,\texttt{beta PROD[Z[[3]], Z[[2]]]} - \texttt{PROD[POWER[Z[[3]], 2], Z[[1]]]} -\\
&\texttt{POWER[Z[[1]], 3]} - \texttt{eps Sin[t] POWER[Z[[3]], 3]}.
\end{aligned} \tag{S.4.3}$$

The third pyramid on the right side of F is simplicity itself. From (10.12.138) we see that this pyramid should be the result of applying the replacement rule to the number 0. Hence, this pyramid is `C0`,

$$0 \rightsquigarrow \texttt{C0} = \{0, 0, \cdots, 0\}. \tag{S.4.4}$$

The remaining lines of the code require little comment. Line 20 sets the initial time to 0, and line 21 defines h in such a way that the final value of t will be 2π. Line 22 establishes the parameter values $\beta = .1$ and $\epsilon = 1.5$, which are those for Figure 1.4.9. Line 23 specifies that the design initial condition is

$$z_1(0) = z_1^{d0} = .3,\ z_2(0) = z_2^{d0} = .4,\ z_3(0) = z_3^{d0} = .5 = \sigma, \tag{S.4.5}$$

and consequently

$$\omega = 1/\sigma = 2. \tag{S.4.6}$$

See (10.12.104). Also, it follows from (10.12.103) and (10.12.106) that

$$q(0) = \omega Q(0) = \omega z_1(0) = (2)(.3) = .6, \tag{S.4.7}$$

$$q'(0) = \omega^2 \dot{Q}(0) = \omega^2 z_2(0) = (2^2)(.4) = 1.6. \tag{S.4.8}$$

Next, lines 24 through 28 specify that the expansion is to be carried out about the initial conditions (7.124). Finally, line 29 invokes the `RK4` code given by (3.6). That is, as before, *no* modifications are required in the integration code.

A few more comments about the output are appropriate. Line 32 shows that the final time t is indeed 2π, as desired. The remaining output lines display the three pyramids that specify the final value of `Zvar`. From the first entry in each pyramid we see that

$$z_1(2\pi) = -0.0493158, \tag{S.4.9}$$

$$z_2(2\pi) = 0.439713, \tag{S.4.10}$$

$$z_3(2\pi) = .5, \tag{S.4.11}$$

when there are no deviations in the initial conditions. The remaining entries in the pyramids are the coefficients in the Taylor series that describe the changes in the final conditions that occur when changes are made in the initial conditions (including the parameter σ). We are, of course, particularly interested in the first two pyramids. The third pyramid has entries only in the first place and the fourth place, and these entries are the same as those in the third pyramid pyramid for `Zvar` at the start of the integration, namely those in `zd0[3] C1 + X[3]`. The fact that the third pyramid in `Zvar` remains constant is the expected consequence of (10.12.138).

At this point we should also describe how the $\mathcal{M}_8$ employed in Section 22.12 was actually computed. It could have been computed by setting $p = 8$ in (4.1) and specifying a small step size h and a great number of steps ns to insure good accuracy. Of course, when $p = 8$, the pyramids are large. Therefore, one does not usually print them out, but rather writes them to files or sends them directly to other programs for further use.

However, rather than using RK4 in (4.1), we replaced it with an adaptive 4-5th order Runge-Kutta-Fehlberg routine that dynamically adjusts the time step h during the course of integration to achieve a specified local accuracy, and we required that the error at each step be no larger than 10^{-12}. (Recall Subsection 2.1.1.) Like the RK4 routine, the Runge-Kutta-Fehlberg routine, when implemented in *Mathematica*, has the property that it can integrate any number of equations both in scalar variable and pyramid form without any changes in the code.[2]

S.5 Relation to the Complete Variational Equations

At this point it may not be obvious to the reader that the use of pyramids in integration routines to obtain Taylor expansions is the same as integrating the complete variational equations. We now show that the integration of pyramid equations is equivalent to the forward integration of the complete variational equations. For simplicity, we will examine the single variable case with no parameter dependence. The reader who has mastered this case should be able to generalize the results obtained to the general case.

[2] A *Mathematica* version of this code is available from Dobrin Kaltchev (kaltchev@triumf.ca) upon request.

In the single variable case with no parameter dependence (1.1) becomes

$$\dot{z} = f(z,t). \tag{S.5.1}$$

Let $z^d(t)$ be some design solution and introduce a deviation variable ζ by writing

$$z = z^d + \zeta. \tag{S.5.2}$$

Then the equation of motion (5.1) takes the form

$$\dot{z}^d + \dot{\zeta} = f(z^d + \zeta, t). \tag{S.5.3}$$

Also, the relations (10.12.14) and (10.12.15) take the form

$$f(z^d + \zeta, t) = f(z^d, t) + g(z^d, t, \zeta) \tag{S.5.4}$$

where g has an expansion of the form

$$g(z^d, t, \zeta) = \sum_{j=1}^{\infty} g^j(t)\zeta^j. \tag{S.5.5}$$

Finally, (10.12.16) and (10.12.17) become

$$\dot{z}^d = f(z^d, t), \tag{S.5.6}$$

$$\dot{\zeta} = g(z^d, t, \zeta) = \sum_{j=1}^{\infty} g^j(t)\zeta^j, \tag{S.5.7}$$

and (10.12.18) becomes

$$\zeta = \sum_{j=1}^{\infty} h^j(t)(\zeta_i)^j. \tag{S.5.8}$$

Insertion of (5.8) into both sides of (5.7) and equating like powers of ζ_i now yields the set of differential equations

$$\dot{h}^{j''}(t) = \sum_{j=1}^{\infty} g^j(t) U_j^{j''}(h^s) \text{ with } j, j'' \geq 1 \tag{S.5.9}$$

where the (universal) functions $U_j^{j''}(h^s)$ are given by the relations

$$\left(\sum_{j'=1}^{\infty} h^{j'}(\zeta_i)^{j'}\right)^j = \sum_{j''=1}^{\infty} U_j^{j''}(h^s)(\zeta_i)^{j''}. \tag{S.5.10}$$

The equations (5.6) and (5.9) are to be integrated from $t = t^{\text{in}} = t^0$ to $t = t^{\text{fin}}$ with the initial conditions

$$z^d(t^0) = z^{d0}, \tag{S.5.11}$$

$$h^1(t^0) = 1, \tag{S.5.12}$$

$$h^{j''}(t^0) = 0 \text{ for } j'' > 1. \tag{S.5.13}$$

Let us now consider the numerical integration of pyramids. Upon some reflection, we see that the numerical integration of pyramids is equivalent to finding the numerical solution to a differential equation with pyramid arguments. For example, in the single-variable case, let $\texttt{Zvar}(t)$ be the pyramid appearing in the integration process. Then, its integration is equivalent to solving numerically the pyramid differential equation

$$(d/dt)\texttt{Zvar}(t) = \texttt{F}(\texttt{Zvar}, t). \tag{S.5.14}$$

We now work out the consequences of this observation. By the inverse of the replacement rule, we may associate a Taylor series with the pyramid $\texttt{Zvar}(t)$ by writing

$$\texttt{Zvar}(t) \rightsquigarrow c_0(t) + \sum_{j\geq 1} c_j(t)x^j. \tag{S.5.15}$$

By (5.15) it is intended that the entries in the pyramid $\texttt{Zvar}(t)$ be used to construct a corresponding Taylor series with variable x. In view of (3.15), there are the initial conditions

$$c_0(t_0) = z^d(t_0), \tag{S.5.16}$$

$$c_1(t_0) = 1, \tag{S.5.17}$$

$$c_j(t_0) = 0 \text{ for } j > 1. \tag{S.5.18}$$

We next seek the differential equations that determine the time evolution of the $c_j(t)$. Under the inverse replacement rule, there is also the correspondence

$$(d/dt)\texttt{Zvar}(t) \rightsquigarrow \dot{c}_0(t) + \sum_{j\geq 1} \dot{c}_j(t)x^j. \tag{S.5.19}$$

We have found a representation for the left side of (5.14). We need to do the same for the right side. That is, we need the Taylor series associated with the pyramid $\texttt{F}(\texttt{Zvar}, t)$. By the inverse replacement rule, it will be given by the relation

$$\texttt{F}(\texttt{Zvar}, t) \rightsquigarrow f(\sum_{j\geq 0} c_j(t)x^j, t). \tag{S.5.20}$$

Here it is understood that the right side of (5.20) is to be expanded in a Taylor series about $x = 0$. From (5.4), (5.5), and (5.10) we have the relations

$$\begin{aligned} f(\sum_{j\geq 0} c_j(t)x^j, t) &= f(c_0(t)) + g(c_0(t), t, \sum_{j\geq 1} c_j(t)x^j) \\ &= f(c_0(t)) + \sum_{k\geq 1} g^k(t)(\sum_{j\geq 1} c_j(t)x^j))^k \\ &= f(c_0(t)) + \sum_{k\geq 1} g^k(t) \sum_{j\geq 1} U_k^j(c_\ell)x^j. \end{aligned} \tag{S.5.21}$$

Therefore, there is the inverse replacement rule

$$\mathtt{F(Zvar},t) \rightsquigarrow f(c_0(t)) + \sum_{k\geq 1} g^k(t) \sum_{j\geq 1} U_k^j(c_\ell)x^j. \tag{S.5.22}$$

Upon comparing like powers of x in (5.19) and (5.22), we see that the pyramid differential equation (5.14) is equivalent to the set of differential equations

$$\dot{c}_0(t) = f(c_0(t)), \tag{S.5.23}$$

$$\dot{c}_j(t) = \sum_{k\geq 1} g^k(t) U_k^j(c_\ell). \tag{S.5.24}$$

Finally, compare the initial conditions (5.11) through (5.13) with the initial conditions (5.16) through (5.18), and compare the differential equations (5.6) and (5.9) with the differential equations (5.23) and (5.24). We conclude that that there must be the relations

$$c_0(t) = z^d(t), \tag{S.5.25}$$

$$c_j(t) = h^j(t) \text{ for } j \geq 1. \tag{S.5.26}$$

We have verified, in the single variable case, that the use of pyramids in integration routines is equivalent to the solution of the complete variational equations using forward integration. As stated earlier, verification of the analogous m-variable result is left to the reader.

We also observe the wonderful convenience that, when pyramid operations are implemented and employed, it is not necessary to explicitly work out the forcing terms $g_a^r(t)$ of Subsection 10.12.1 and the universal functions $U_r^{r''}(h_n^s)$ of Subsection 10.12.3, nor is it necessary to explicitly set up the complete variational equations (10.12.36). All these complications are handled implicitly and automatically by the pyramid routines.

Exercises

S.5.1. Verify, in the general m variable case, that the use of pyramids in integration routines is equivalent to the solution of the complete variational equations using forward integration.

Bibliography

[1] R. Neidinger, “Computing Multivariable Taylor Series to Arbitrary Order”, *Proc. of Intern. Conf. on Applied programming languages*, San Antonio, pp. 134-144 (1995).

[2] Wolfram Research, Inc., *Mathematica*, Version 7.0, Champaign, IL (2008).

[3] Dobrin Kaltchev (*TRIUMF, 4004 Wesbrook Mall, Vancouver, B.C., Canada V6T 2A3*) designed and wrote all the *Mathematica* code for this appendix, and he and Alex Dragt coauthored the text. D. Kaltchev wishes to thank his colleagues from TRIUMF and CERN, especially Richard Abram Baartman, for their interest and support.

[4] D. Kalman and R. Lindell, “A recursive approach to multivariate automatic differentiation”, *Optimization Methods and Software*, Volume 6, Issue 3, pp. 161-192 (1995).

[5] M. Berz, “Differential algebraic description of beam dynamics to very high orders”, *Particle Accelerators* 24, p. 109 (1989).