An Algorithmic Study of the Construction of Higher-order One-dimensional Castillo-Grone Mimetic Gradient and Divergence Operators

Eduardo J. Sanchez and Jose E. Castillo

December 13, 2013

Publication Number: CSRCR2013-02

Computational Science & Engineering Faculty and Students Research Articles

Database Powered by the Computational Science Research Center Computing Group & Visualization Lab

COMPUTATIONAL SCIENCE & ENGINEERING

SAN DIEGO STATE UNIVERSITY

Computational Science Research Center
College of Sciences
5500 Campanile Drive
San Diego, CA 92182-1245
(619) 594-3430

An Algorithmic Study of the Construction of Higher-order One-dimensional Castillo-Grone Mimetic Gradient and Divergence Operators

Eduardo J. Sanchez and Jose E. Castillo
esanchez@mail.sdsu.edu and jcastillo@mail.sdsu.edu
Computational Science Research Center
San Diego State University

Table of Contents

1. Introduction.

2. An algorithm for the Castillo-Runyan Method.
2.1. Approximation at the interior cells.
2.2. Approximation at the boundary points.
2.3. Assembling the final matrix.

3. Concluding remarks and directions for future work.

References.

Observation

This entire document is written in **Maple 16**; its related PDF file will be generated depicting the construction of a 4th-order accurate mimetic divergence, thus matching Appendix J of [3]. However, the interested reader can download this Maple worksheet, make necessary changes, and produce the appropriate tailored results:

restart;

Please state the required order of accuracy (an even integer number, greater or equal than 2):

$k := 4;$

$$4 \quad \textbf{(2.1)}$$

Please state whether you want a gradient (0) or a divergence (1):

$p := 1;$

$$1 \quad \textbf{(2.2)}$$

Thanks.

1. Introduction

In this work, we present an algorithm for the construction of higher-order one-dimensional Castillo-Grone mimetic gradient and divergence operators. The creation of this algorithm is motivated by the development of an Applications Programming Interface for the implementation of Mimetic Finite

Differences. Section 2 presents each section of the algorithm, as well as the results they yield, given an even order of accuracy *k*. Section 3 presents a set of concluding remarks, as well a reference to the existing subsequent research.

In 2003, Castillo and Grone [1] explained the construction of higher-order mimetic gradient and divergence operators that satisfy a discrete and **extended form of Gauss' Divergence Theorem**. This explanation is referred to as the **Castillo—Grone Method**. Runyan and Castillo [2] presented a similar approach for constructing mimetic operators. This second explanation is referred to as the **Castillo—Runyan Method (CRM)**. Although very descriptive, these works lack the definition of an algorithm that can be used, directly and unambiguously, to automate the construction of mimetic operators of any desired (even) order of accuracy. Furthermore, both works lack the description of their respective methodologies to construct gradient operators, since they only exemplify the construction of divergence operators. Both methods are condensed in [3].

In this work, we present an algorithm to construct higher-order mimetic gradient and divergence operators. We start by generalizing the CRM, thus creating the Castillo-Runyan Algorithm for the construction of higher-order mimetic gradient and divergence operators. We take the liberty to make minor improvements upon the original methodology presented in [2] (see Section 2). We also explicitly explain the restrictions of the CRM when dealing with higher-order mimetic operators. In Section 3, we explain how these restrictions motivate a modification of the CRM that approaches the problem of constructing higher-order mimetic operators from the perspective of constrained linear optimization, which is the focus of [4].

2. An Algorithm for the Castillo-Runyan Method

The construction of a mimetic differential operator by means of Mimetic Finite Differences (MFDs) borrows some ideas from Standard Finite Differences as implemented on a staggered grid-also called Staggered Finite Differences (SFDs). Specifically, the construction of the mimetic operator—approximating the derivative at the interior cells of a 1-D uniform staggered grid—is the same as in SFDs. The difference arises in the treatment of the boundaries.

2.1. Approximation at the interior points

Mimetic differential operators approximate their continuous counterparts, with given an even order of accuracy:

```
with(LinearAlgebra):
interface(rtablesize = k*k):
k;
```

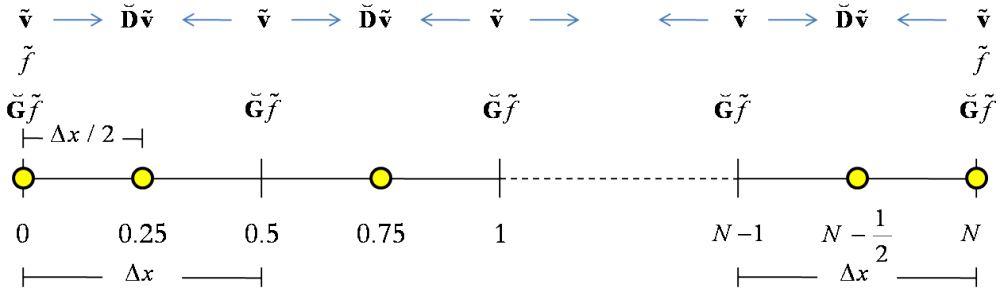
4

Observation: In the previous code snippet, the Maple interface was configured to always display structures of a size proportional to the order of accuracy.

The next step is to compute the approximation at the interior. This implies computing a collection of coefficients approximating the derivative using the surrounding neighbors of a given point, or, specifically:

• Node, in the case of the gradient.
• Cell center, in the case of the divergence.

In the following image, nodes are denoted as vertical lines, whereas cell centers are denoted as yellow circles:



Such image also summarizes how the results of these mimetic operators are bound to this grid.

Taylor-expanding around a given point on our staggered grid, and solving for the coefficients that approximate the first-order derivative as both the gradient and the divergence intend to, can be achieve by means of **Vandermonde matrices**, and what we call an **order-selector vector**. In order for us to generate the correct Vandermonde matrix, we need to generate its **generator vector**, which will be constructed based on the spatial coordinates of the neighboring points we intend to consider in the Taylor-expansion, for the current point of interest. The generator vector is created as follows:

```
i := 1:
g := Vector(k):
for j from (1/2 - k/2) to (k/2) do
  g[i] := j:
  i := i + 1:
end do:
Transpose(g);
```

$$\begin{bmatrix} -\frac{3}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{3}{2} \end{bmatrix}$$

As it can be seen, the latter result gathers the coordinates of the neighbors of the point we intend to Taylor-expand on. The related Vandermonde matrix, along with the order-selector vector required to compute the values of the coefficients approximating the first-order derivative, with a k-th order of accuracy, is constructed as follows:

```
T := Transpose(VandermondeMatrix(g));
o := Vector(k):
o[2] := 1:
o;
s := LinearSolve(T, o):
Transpose(s);
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -\frac{3}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{3}{2} \\ \frac{9}{4} & \frac{1}{4} & \frac{1}{4} & \frac{9}{4} \\ -\frac{27}{8} & -\frac{1}{8} & \frac{1}{8} & \frac{27}{8} \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{24} & -\frac{9}{8} & \frac{9}{8} & -\frac{1}{24} \end{bmatrix}$$

These three results include: the generated Vandermonde matrix, the form of the order-selector vector, and the computed set of approximating coefficients, which have been gathered within a **stencil vector**.

2.2. Approximation at the boundary points

The main difference between the Mimetic Finite Differences (MFDs) and Standard Staggered Finite Differences (SFDs) approaches is the treatment of the boundaries. This must be the case, since we intend for our mimetic operator to comply with the following extended form of Gauss' Divergence Theorem:

$$\int_{\Omega} (\nabla \cdot \mathbf{v}) f \, \mathrm{d}V + \int_{\Omega} (\nabla f) \mathbf{v} \, \mathrm{d}V = \int_{\partial\Omega} f (\mathbf{v} \cdot \mathbf{n}) \, \mathrm{d}S.$$

Such an expression, when considered in a one-dimensional domain, boils down to the the following result, for $\Omega = [a, b]$:

$$\int_a^b \frac{\mathrm{d}v}{\mathrm{d}x} f \, \mathrm{d}x + \int_a^b \frac{\mathrm{d}f}{\mathrm{d}x} v \, \mathrm{d}x = \mathrm{v}(b) f(b) - v(a) f(a).$$

From the latter, we claim that a given discrete differential operator is **mimetic** if and only if it complies with the following conditions:

1. For the gradient ($\forall\ x \in \Omega : v(x) = 1$), we require:

$$\int_{\Omega} \nabla f \, dV = \int_{a}^{b} \frac{df}{dx}\, dx = f(b) - f(a).$$

2. For the divergence ($\forall\ x \in \Omega : f(x) = 1$), we require:

$$\int_{\Omega} (\nabla \cdot \mathbf{v})\, dV = \int_{a}^{b} \frac{dv}{dx}\, dx = v(b) - v(a).$$

We will refer to the previously stated conditions as the "**mimetic conditions**" for each operator.

Based on a decision parameter $p \in \{0, 1\}$, we will compute either a gradient ($p = 0$) or a divergence ($p = 1$):

```
p;
```

1

Since we are interested in approximating a derivative at the boundaries, we must start by computing the Vandermonde matrices at these points. Clearly, this process depends on whether we are interested in either a gradient or a divergence, because they are evaluated at different points within the staggered grid. As in the previous section, we start by defining a generator vector:

```
g := Vector((3/2)*k):
if (p = 0) then
  g[1] := 0:
  i := 2:
  for j from (1/2) by 1 to ((3/2)*k - 1) do
    g[i] := j;
    i := i + 1;
  end do:
else
  i := 1:
  for j from -1/2 by 1 to ((3/2)*k - 1) do
    g[i] := j:
    i := i + 1:
  end do:
end if:
Transpose(g);
```

$$\begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{3}{2} & \frac{5}{2} & \frac{7}{2} & \frac{9}{2} \end{bmatrix}$$

The previous vector is the generator vector for the first Vandermonde matrix approximating the derivative at the first boundary point. As it can be noticed, we must compute $\frac{3}{2}k$ approximations, as stated by the CRM. In the algorithm we want to build, we will only construct as many as the dimension of the kernel of the first Vandermonde matrix approximating the points at the boundary. This number depends on the operator we want to built, and it is computed as follows [3]:

```
if (p = 0) then
  d := (k/2);
else
  d := (k/2) - 1;
end if;
```

1

Observation: The previous snippet of code should make it apparent to the reader that every section of this algorithm should not rely on a conditional, but on including the parameter p in their description. For example, in the previous snippet of code, d could be defined as $\left(\frac{k}{2} - p\right)$. However, since the creation of this algorithm is motivated by its inclussion on an API that implements it, we believe it is important to maintain the instructions for the construction of each operator separated, so that we can implement them in this API, in different classes.

Once we have computed the dimension of the kernel of the first Vandermonde matrix approximating the derivative at the boundary, we proceed to compute them all. However, since we are not interested in storing such matrices, once we compute each Vandermonde matrix, we will define a fitting order-selector vector, that would allow us to compute the desired stencil. The first Vandermonde matrix will be computed using the previously built generator vector:

```
Transpose(g);
```

$$\begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{3}{2} & \frac{5}{2} & \frac{7}{2} & \frac{9}{2} \end{bmatrix}$$

Since the successive Vandermonde matrix intends to approximate the derivative at the next boundary point, it is the clear that we should shift the entrances of the aforementioned generator vector by 1. Doing so will yield the appropriate generator vector for the next Vandermonde matrix approximating the derivative at the immediately adjacent boundary point. In summary, per each one of the d boundary points, we will generate a Vandermonde matrix, use it to solve for the stencil, and then shift the entrances of **g** by one, thus paving the way for the next iteration. Such approach is implemented in the following code snippets:

• Generate the order-selector vector **o** and create a copy of the initial generator vector **g**:

```
o := Vector(k + 1):
o[2] := 1:
o;

h := g:
Transpose(h);
```

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{3}{2} & \frac{5}{2} & \frac{7}{2} & \frac{9}{2} \end{bmatrix}$$

- Create the array of Vandermonde matrices ($\mathbb{V}$) just to see them, but these are not required to be stored. We store the computed set of "pseudo rows" (`r`) of the desired operator. However, as explained in both [2] and in [3], these solutions will not be unique, but they will depend on a linear combination of the vector of the kernel of each one of the created Vandermonde matrices. Based on this, we will instruct the Maple solver to label these parameters as a each. We can then store an evaluation of each computed row, assuming $a = 0$. After we have stored all of these computed "**pseudo rows**", we will proceed to explicitly compute the kernel of the first Vandermonde matrix. The loop solving each pseudo row now follows:

```
V := Array(1..d):
r := Array(1..d):

# Compute the near-the-boundary rows of the PI matrix:
for j from 1 by 1 to d do
  # Compute the j-th Vandermonde matrix:
  V[j] := Transpose(VandermondeMatrix(h, (3/2)*k, k + 1)):

  # Generate AND solve the Vandermonde system for the current row:
  r[j] := LinearSolve(V[j], o, free = a):

  # Remove the presence of the parametric portion of the solution:
  r[j] := eval(r[j], a = 0):

  # Shift entrances of generator computing the next Vandermonde matrix:
  for z from 1 by 1 to (3/2)*k do
    h[z] := h[z] - 1:
  end do:
end do:
V;
o;
r;
```

$$\left[\left[\begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 & 1 \\ -\frac{1}{2} & \frac{1}{2} & \frac{3}{2} & \frac{5}{2} & \frac{7}{2} & \frac{9}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{9}{4} & \frac{25}{4} & \frac{49}{4} & \frac{81}{4} \\ -\frac{1}{8} & \frac{1}{8} & \frac{27}{8} & \frac{125}{8} & \frac{343}{8} & \frac{729}{8} \\ \frac{1}{16} & \frac{1}{16} & \frac{81}{16} & \frac{625}{16} & \frac{2401}{16} & \frac{6561}{16} \end{array}\right]\right]$$

$$\left[\begin{array}{c} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{array}\right]$$

$$\left[\left[\begin{array}{c} -\frac{11}{12} \\ \frac{17}{24} \\ \frac{3}{8} \\ -\frac{5}{24} \\ \frac{1}{24} \\ 0 \end{array}\right]\right]$$

The previously computed set of pseudo rows are particular instances of the generalized solution provided by Maple, in terms of a linear combination of the vector of the kernel of the matrices, as we already explained. We must now generate this kernel. However, given the nature of the d Vandermonde matrices stored in the $\mathbb{V}$ array, they will all posses the same kernel. Therefore, we impose the convention of computing the kernel of the first Vandermonde matrix:

```
N := NullSpace(V[1]);
```

$$\left\{\left[\begin{array}{c} -1 \\ 5 \\ -10 \\ 10 \\ -5 \\ 1 \end{array}\right]\right\}$$

At this point we already have a collection of pseudo rows and the elements of the kernel that will eventually define the real rows of our matrix operator. However, we need to impose the respective mimetic condition for the operator being computed. For this we pay attention to the discrete analog of the previously discussed continuous mimetic conditions, as follows:

• In the case of the gradient, we have, in the continuous world, the following requirement:

$$\int_{\Omega} \nabla f \, dV = \int_{a}^{b} \frac{df}{dx} \, dx = f(b) - f(a).$$

In order for us to understand the discrete analog of such condition, we must introduce the following notation:

(1) Let h be the discrete counterpart of dx. Let N be the number of cells in our staggered grid, caused by the selection of h.
(2) Let $\mathbf{e} = [1,..., 1] \in \mathbb{R}^k$.
(3) let $\mathbf{G}$ and $\mathbf{f}$ be the discrete gradient operator we intend to compute, and the discretized scalar-valued quantity we want to compute it for, respectively. The discrete counterpart of the

mimetic condition for the gradient becomes:

$$\langle \mathbf{Gf}, h\mathbf{e} \rangle = f_N - f_0 = \langle (-1, 1), (f_N, f_0) \rangle.$$

• Analogously, we can state the following discrete counterpart of the mimetic condition for the divergence operator:

$$\langle \mathbf{Df}, h\mathbf{e} \rangle = v_N - v_0.$$

If we consider, for example, that $\langle \mathbf{Dv}, h\mathbf{e} \rangle = h \langle \mathbf{Dv}, \mathbf{e} \rangle$, then our goal is to compute the rows of the divergence such that: $\langle \mathbf{D}, \mathbf{e} \rangle = [-1, 0, ...0, 1]^T$. As suggested in both [2], and in [3], we focus on the set of rows of **D**, which can be thought of as the total set of rows of a submatrix **A** to **D**. Since $\langle \mathbf{D}, \mathbf{e} \rangle = \langle \mathbf{e}^T, \mathbf{D} \rangle$, we are then forced to request that the column sum of our desired operator equal -1, for the first column, 1 for the last column, and 0 everywhere else.

However, when assembling such system of equations, we wee that such system has no solution [1], [3]. The same argument can be directly applied to the gradient operator, so we are forced to consider weighted inner products, instead of the standard inner products we choose to describe the discrete analogs of the continuous mimetic conditions. Based on this, we consider the following mimetic conditions:

• For the gradient:

$$\langle \mathbf{Gf}, h\mathbf{e} \rangle_{\mathbf{P}} = \langle \mathbf{PGf}, h\mathbf{e} \rangle = f_N - f_0.$$

• For the divergence:

$$\langle \mathbf{Dv}, h\mathbf{e} \rangle_{\mathbf{Q}} = \langle \mathbf{QDv}, h\mathbf{e} \rangle = v_N - v_0.$$

Where **P** and **Q** are strictly diagonal, positive-definite weighing matrices. When considering the submatrix approach previously explained, our problem is now slightly different: we must now compute the correct weights that would allow us to impose the mimetic conditions required in each operator. And, once again, these should be strictly positive.

Considering the sub matrix approach, forces us to first consider the dimension of this submatrix we want to compute. Again, let **A** be this matrix, then:

$$\mathbf{D}(\mathbf{A}) = \begin{bmatrix} & & 0 & \cdots & & & & & & & \cdots & 0 \\ & \mathbf{A} & 0 & \cdots & & & & & & & \cdots & 0 \\ & & 0 & \cdots & & & & & & & \cdots & 0 \\ 0 & \cdots & 0 & s_1 & s_2 & \cdots & s_k & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \ddots & \ddots & \cdots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & s_1 & s_2 & \cdots & s_k & 0 & \cdots & 0 \\ 0 & \cdots & & & & & & \cdots & 0 & & & \\ 0 & \cdots & & & & & & \cdots & 0 & & \mathbf{A}' & \\ 0 & \cdots & & & & & & \cdots & 0 & & & \end{bmatrix}$$

where $\{s_i\}_{i=1}^{i=k}$ denote the values contained in the stencil vector computed for the interior cells.

According to [3], $\mathbf{A} \in \mathbb{R}^{N \times \frac{3}{2}N}$. Therefore, we must assemble a system of equations to compute these weights, in which we consider our pseudo rows, and some extra information that would allows us to make up for the remaining $\frac{3}{2}k$ rows. Let $\mathbf{\Pi}$ denote the matrix within this system, which intends to include the pseudo rows, as well as this extra information. It turns out that this extra information is given by the values computed for the interior cell; that is, the elements of $\mathbf{s}$. In the original CRM, the loop computed the pseudo rows, would not stop after $\dim\left(\ker\left(\mathrm{V}_1\right)\right)$ or d iterations, but after $\frac{3}{2}k$ iterations, yielding, as pseudo rows, instances of $\mathbf{s}$. Therefore, we use these already known values, and we extend our matrix to built with the information from the kernel we have already computed. In summary, the construction of this matrix, is done as follows:

```
PI := Matrix((3/2)*k, 0, datatype = float[8]):

if (p = 0) then
  # Add the columns computed from the boundary rows:
  for i from 1 to (k/2) do
    PI := <r[(k/2) - (i - 1)]|PI>;
  end do:

  # Add the elements from the stencil:
  xx := 1:
  yy := -1:
  for j from (k/2) to (k + yy) do
    a := ZeroVector(j - (k/2) + xx);
    b := ZeroVector(k - j + yy);
    a := <a,s,b>;
    PI := <PI|a>;
  end do:

  # Complete the construction with the elements from the kernel:
  for i from 1 by 1 to (k/2 - 1) do
    PI := <PI|N[(k/2) - i]>;
  end do:
else
  # Add the columns computed from the boundary rows:
  for i from 1 by 1 to (k/2) - 1 do
    PI := <r[((k/2) - 1) - (i - 1)]|PI>:
  end do:

  # Add the elements from the stencil:
  for j from (k/2) by 1 to k do
    a := ZeroVector(j - (k/2)):
    b := ZeroVector(k - j):
    a := <a,s,b>:
    PI := <PI|a>:
  end do:

  # Complete the construction with the generators:
  for i from 1 by 1 to (k/2) - 1 do
    PI := <PI|N[i]>:
  end do:
end if:
simplify(PI);
```

$$\begin{bmatrix} -\frac{11}{12} & \frac{1}{24} & 0 & 0 & -1 \\ \frac{17}{24} & -\frac{9}{8} & \frac{1}{24} & 0 & 5 \\ \frac{3}{8} & \frac{9}{8} & -\frac{9}{8} & \frac{1}{24} & -10 \\ -\frac{5}{24} & -\frac{1}{24} & \frac{9}{8} & -\frac{9}{8} & 10 \\ \frac{1}{24} & 0 & -\frac{1}{24} & \frac{9}{8} & -5 \\ 0 & 0 & 0 & -\frac{1}{24} & 1 \end{bmatrix}$$

The treatment of the previous matrix, when higher orders are considered, will be the cornerstone of a proposed refinement to both the CGM and the CRM [4]. For now, let us concentrate on the right-hand side that is required to complete the computation of the weights. Since we are thinking about computing the rows of a sub-matrix to our operator, we want to do it in a way that the column sum of this submatrix equals those values, such that, when inserted into the final operator, they will fulfill the mimetic condition: $\langle \mathbf{D}, \mathbf{e} \rangle = [-1, 0, ...0, 1]^T$. We construct such vector, using the values of the stencil vector, as follows:

```
h := Vector((3/2)*k):
h[1] := -1:
for i from (k/2 + 2) by 1 to ((3/2)*k) do
  x := 0:
  for j from 1 by 1 to (i - (k/2 + 1)) do
    x := x + s[j]:
  end do:
  h[i] := -1*x:
end do:
h;
```

$$\begin{bmatrix} -1 \\ 0 \\ 0 \\ -\frac{1}{24} \\ \frac{13}{12} \\ -\frac{1}{24} \end{bmatrix}$$

A symbolic depiction of the system to solve can be easily rendered through the following snippet of code:

```
MatrixVectorMultiply(PI,Vector((3/2)*k - 1,symbol='q')) = h;
```

$$\begin{bmatrix} -\frac{11}{12} q_1 + \frac{1}{24} q_2 - q_5 \\ \frac{17}{24} q_1 - \frac{9}{8} q_2 + \frac{1}{24} q_3 + 5 q_5 \\ \frac{3}{8} q_1 + \frac{9}{8} q_2 - \frac{9}{8} q_3 + \frac{1}{24} q_4 - 10 q_5 \\ -\frac{5}{24} q_1 - \frac{1}{24} q_2 + \frac{9}{8} q_3 - \frac{9}{8} q_4 + 10 q_5 \\ \frac{1}{24} q_1 - \frac{1}{24} q_3 + \frac{9}{8} q_4 - 5 q_5 \\ -\frac{1}{24} q_4 + q_5 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ -\frac{1}{24} \\ \frac{13}{12} \\ -\frac{1}{24} \end{bmatrix}$$

In the end, computing the weights can be done as follows:

```
P := simplify(LinearSolve(PI, h));
```

$$\begin{bmatrix} \frac{649}{576} \\ \frac{143}{192} \\ \frac{75}{64} \\ \frac{551}{576} \\ -\frac{25}{13824} \end{bmatrix}$$

At this point, we must pay attention to a very important issue: It is nothing but mere fortune that all weights are strictly positive, but that is just the case until we reach 8th-order accuracy for the divergence, and 10-th order accuracy for the gradient. When we reach these orders of accuracy, negative weights start to arise.

2.3. Assembling the final matrix

The final step in the algorithm is to compute the final matrix implementing the operator. For any application of this algorithm, it is better not to compute a matrix, but merely the collection of values that unequivocally define the approximation coefficients at the boundaries, and at the interior of the discretized domain. However, just to give a completion to this document, we

will assemble a matrix-formatted discrete differential operator with a minimum size so that the values at the boundaries do not overlap.

The first step is to extract the scalars that have been computed within the weights vector:

```
lambda := Vector(k/2 - 1):
for i from 1 by 1 to (k/2) - 1 do
  lambda[i] := P[(k + (k/2 - 1)) - (i - 1)]:
end do:
lambda;
```

$$\left[-\frac{25}{13824} \right]$$

The final step is to compute the individual rows of the sub-matrix **A**, which contain the special rows treating the boundaries, within the objective operator. This final step is divided into two parts:

1. Solve for the α-values:

```
alpha := Vector(k/2 - 1):
for i from 1 to (k/2 - 1) do
  alpha(i) := lambda(i)/P(i):
end do:
alpha;
```

$$\left[-\frac{25}{15576} \right]$$

2. Use the α-values to solve for each row within the sub-matrix, using the **Π** matrix as the starting point:

```
A := DeleteColumn(PI,[(3/2)*k - (k/2 - 1)..(3/2)*k])^%T:
for i from 1 to (k/2 - 1) do
  for j from 1 to (3/2)*k do
    A(i,j) := A(i,j) + alpha(i)*N[i][j]:
  end do:
end do:
A;
evalf(A);
```

$$\begin{bmatrix} -\frac{4751}{5192} & \frac{909}{1298} & \frac{6091}{15576} & -\frac{1165}{5192} & \frac{129}{2596} & -\frac{25}{15576} \\ \frac{1}{24} & -\frac{9}{8} & \frac{9}{8} & -\frac{1}{24} & 0 & 0 \\ 0 & \frac{1}{24} & -\frac{9}{8} & \frac{9}{8} & -\frac{1}{24} & 0 \\ 0 & 0 & \frac{1}{24} & -\frac{9}{8} & \frac{9}{8} & -\frac{1}{24} \end{bmatrix}$$

```
[[-0.9150616333, 0.7003081664, 0.3910503338, -0.2243836672, 0.04969183359,
    -0.001605033385],
  [0.04166666667, -1.125000000, 1.125000000, -0.04166666667, 0., 0.],
  [0., 0.04166666667, -1.125000000, 1.125000000, -0.04166666667, 0.],
  [0., 0., 0.04166666667, -1.125000000, 1.125000000, -0.04166666667]]
```

Once the sub-matrix **A** has been created, it is just of matter of applying the proper permutation to it, thus creating its correspondent sub-matrix for the east boundary, which will complete our required operator, since every instance of the operator will possess both matrices, as well as iterated instances of the stencil vector, all divided by the proper scaling, based on the grid chosen step size for the discretization.

3. Concluding remarks and directions for future work

In this work, we have presented a general algorithm implementing the Castillo-Runyan Method for the construction of higher-order mimetic differential operators. This algorithm allows for the construction of any operator (Divergence and Gradient), starting from a given even order of accuracy, *k*. The added value to this report, is the fact that it has been written in Maple 16. Therefore, every single typed result comes from the execution of this very same Maple work sheet, thus validating the approach hereby presented.

Future work will be based on presenting a modification to this method, to explicitly address the constraint that $\mathbf{q} > 0$. That is, to explicitly impose the constraint that all of the computed weights should be positive-definite. In **[1]**, it is stated that an scheme to compute "optimal weights" is yet to be studied, thus unveiling the consideration of the existence of "optimal weights".

In **[4]**, the authors tackle this problem by presenting a modification to the CRM based on constrained linear optimization. This modification yields a CLO-based algorithm, where the weights are computed as the solution of a residual minimization linear constrained optimization problem. Such approach allows to use the Simplex Method, to solve for such weights, while explicitly taking into the account the aforesaid constraint.

References

[1] Castillo, J. and Grone, R. (2003). *A matrix analysis approach to higher-order approximations for divergence and gradients satisfying a global conservation law.* Siam J. Matrix Anal. Appl., 25:128–142.

[2] Runyan, J. (2011). *A novel higher order finite difference time domain method based on the*

Castillo-Grone mimetic curl operator with application concerning the time-dependent Maxwell equations. Master's thesis, San Diego State University, San Diego, CA.

[3] Castillo, J. and Miranda, G. (2013). *Mimetic Discretization Methods.* 260 pages. CRC Press (Boca Raton, FL), 1st edition. Referenced 4 times.

[4] Sanchez, E., Blomgren, P. and Castillo, J. (2013). *On the role of constrained linear optimization to construct higher-order mimetic divergence operators*. Journal of Computational and Applied Mathematics. Pending peer review.