EXPLICIT EXPRESSION AND FAST ALGORITHMS FOR THE INVERSE OF SOME MATRICES ARISING FROM IMPLICIT TIME INTEGRATION

SHISHUN LI* AND HUILE WEI†

Abstract. In this paper, we first present an explicit expression for the inverse of a type of matrices. As special applications, the inverse of some matrices arising from implicit time integration techniques, such as the well-known implicit Runge-Kutta schemes and block implicit methods, can also be explicitly determined. Adiitionally, we introduce three fast algorithms for computing the elements of the inverse of these matrices in $O(n^2)$ arithmetic operations, i.e., the first one is based on Traub algorithm for fast inversion of Vandermonde matrices, while the other two utilize the special structure of the matrices. Finally, some symbolic and numerical results are presented to show that our algorithms are both highly efficient and accurate.

Key words. Matrix inversion, fast algorithm, Runge-Kutta matrix, implicit time integration,

AMS subject classifications. 15A09, 65F05, 65D30, 68Q25, 65Y20

1. Introduction.

General speaking, it is unnecessary and inadvisable to actually compute matrix inverse in the vast majority of practical computational problems. However, there exist some matrices and their inverse play a very important role in practical applications. In this paper, we investigate a type of matrices with special structure and design some fast algorithms to compute their inverse. Let $c_0, c_1, \cdots, c_n$ be distinct real or complex numbers, we consider the explicit expression of the elements of the matrix

$$\boldsymbol{W} = \boldsymbol{V}^{-1}\boldsymbol{H}\boldsymbol{V} \equiv \begin{pmatrix} w_{11} & \boldsymbol{w}_1^T \\ \boldsymbol{w}_2 & \boldsymbol{W}_n \end{pmatrix}, \tag{1.1}$$

where $w_{11} \in \mathbb{R}$, $\boldsymbol{w}_1, \boldsymbol{w}_2 \in \mathbb{R}^n$, $\boldsymbol{W}_n \in \mathbb{R}^{n\times n}$,

$$\boldsymbol{H} = \begin{pmatrix} 0 & & & \\ 1 & \ddots & & \\ & \ddots & \ddots & \\ & & n & 0 \end{pmatrix} \quad \text{and} \quad \boldsymbol{V} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ c_0 & c_1 & c_2 & \cdots & c_n \\ c_0^2 & c_1^2 & c_2^2 & \cdots & c_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_0^n & c_1^n & c_2^n & \cdots & c_n^n \end{pmatrix}.$$

Here $\boldsymbol{V} \in \mathbb{R}^{(n+1)\times(n+1)}$ is a Vandermonde matrix. It is well-known that the inverse of $\boldsymbol{V}$ can be given explicitly, that is, there exists an explicit formula for the element of $\boldsymbol{V}^{-1}$. By generalizing the scheme for computing the inverse of a Vandermonde matrix described in [26], we give a explicit expression of the matrix $\boldsymbol{W}$, i.e., the elements of $\boldsymbol{W}$ can be computed by the basic four arithmetic operations $(+, -, *, /)$ of c_i.

The implicit Runge-Kutta (IRK) methods and block implicit methods (BIM) have been widely used for the time discretization of ordinary and partial differential equations [6, 4, 10, 17, 16, 18, 21, 22]. The inverse of IRK matrix and BIM matrix are necessary for stability analysis, and plays an important role in the construction of new time discretization methods with desirable properties [2, 17]. In this paper, we

---

*School of Mathematics and Statistics, Xinyang Normal University, Xinyang, 464000, Henan, P.R.China. (lss6@sina.com).

†School of Mathematics and Information Science, Henan Polytechnic University, Jiaozuo, 454000, Henan, P.R.China (whlwhl2002@sina.com).

prove that the inverse of the IRK matrix and BIM matrix is the submatrix of $\boldsymbol{W}^T$, i.e., the inverse of IRK matrix and BIM matrix is exactly $\boldsymbol{W}_n^T$ if c_i $(1 \leq i \leq n)$ are the zeros of the Legendre polynomials and $c_i = i$ $(1 \leq i \leq n)$, respectively. Consider a parabolic equations of the form

$$u_t - \nabla \cdot (\alpha(x)\nabla u) + \beta(x) \cdot \nabla u = f(x,t), \text{ in } \Omega \times (0,T] \tag{1.2}$$

equipped with appropriate initial and boundary conditions, where $\Omega \subset R^d$ $(d = 2 \text{ or } 3)$ is a bounded, open polygonal (or polyhedra) domain, $f(x,t) \in L^2(\Omega \times [0,T])$, $0 < \alpha_0 \leq \alpha(x) \leq \alpha_1 < +\infty$, and all the coefficients are sufficiently smooth. The discretization of (1.2) by IRK in time and finite element in space produces a system of equations with the coefficient matrix

$$\boldsymbol{I} \otimes \boldsymbol{M} + \tau \boldsymbol{A} \otimes \boldsymbol{K}, \tag{1.3}$$

where $\otimes$ denotes the Kronecker product, τ is the time step, $\boldsymbol{I} \in \mathbb{R}^{r\times r}$, $\boldsymbol{A} \in \mathbb{R}^{r\times r}$, $\boldsymbol{M} \in \mathbb{R}^{N\times N}$ and $\boldsymbol{K} \in \mathbb{R}^{N\times N}$ denote identity matrix, the coefficient matrix of r-stage IRK, the mass matrix and the stiffness matrix, respectively. In practice, (1.3) is often large and highly ill-conditioned, and parallel preconditioners are necessary to construct for solving the system with coefficient matrix (1.3). Usually, it is preferable to solve a equivalent system with the coefficient matrix

$$\boldsymbol{A}^{-1} \otimes \boldsymbol{M} + \tau \boldsymbol{I} \otimes \boldsymbol{K} \tag{1.4}$$

since it can significantly reduce the computational cost [1, 3, 5, 10, 19]. Moreover, it requires less memory since $\boldsymbol{A}^{-1} \otimes \boldsymbol{M}$ is in general sparser than $\boldsymbol{A} \otimes \boldsymbol{K}$, and the stiff part of the matrix is block diagonal so that the analysis and solution more tractable [24]. Various preconditioners based on $\boldsymbol{A}^{-1}$ are constructed, such as, $ILU(0)$-based stage-coupled and stage-parallel preconditioners [20], linear and nonlinear parallel preconditioner [25, 24]. and the lower triangular preconditioners [1]. Hence, the computation of $\boldsymbol{A}^{-1}$ is very important for solving the system of equations arising from the discretization of time-dependent PDEs, and we wiil present an explicit formula to compute $\boldsymbol{A}^{-1}$.

The explicit formula for the elements of $\boldsymbol{W}$ is well suited for hand or machine calculation. However, the computation of the $(n+1)^2$ elements requires $O(n^3)$ arithmetic operations, and it is clearly too expensive in compute time for large n. In this paper, we present three fast algorithms to compute all the elements of $\boldsymbol{W}$ in only $O(n^2)$ floating point operations (flops). The explicit formula for Vandermonde matrix was first presented by Traub in [26], and a algorithm with the complexity $6n^2$ flops was also proposed by using its special structure. This algorithm has been generalized to invert the Vandermonde-like matrices [7, 8, 9, 12, 23, 27]. Following the idea in [26], we first present a fast algorithm to compute all the elements of $\boldsymbol{W}$ in only $O(n^2)$ flops. Similar to Traub algorithm for Vandermonde matrix, this algorithm is numerically unstable and inaccurate due to the rapid propagation of roundoff errors. Moreover, the solution components may soon exceed the representable machine number and produce overflow as n increases. Actually, we observe that $\boldsymbol{W}$ has better structure than Vandermonde matrix although it is singular. By using a trick to avoid overflow and underflow in multiplication and division, we derive another two algorithms with $O(n^2)$ complexity. It is shown that our algorithms are both highly efficient and accurate for computing all the elements of $\boldsymbol{W}$.

The remaining part of the paper is organized as follows. In Section 2, we first give a explicit formula for the matrix $\boldsymbol{W}$, then prove that the inverse of IRK matrix and

BIM matrix is exactly the submatrix of $\boldsymbol{W}$. In Section 3, three fast algorithms with the complexity $O(n^2)$ flops are presented to compute the elements of $\boldsymbol{W}$. In Section 4, some symbolic and numerical results are presented to demonstrate the performance of the proposed algorithms. Finally, some concluding remarks are given in Section 5.

2. The explcit formula for the matrix inversion. In this section, we first present the explicit formula for the matrix $\boldsymbol{W}$. Then, we prove that the inverse of the IRK matrix and the BIM matrix is exactly $\boldsymbol{W}_n^T$ when choosing special c_i $(1 \le i \le n)$.

2.1. The explicit formula for the inverse of W **.** Let $\varphi(x)$ be a monic polynomial of degree $n+1$,

$$\varphi(x) = \prod_{k=0}^{n} (x - c_k). \tag{2.1}$$

From the well-known Vieta's formula, we have

$$\varphi(x) = \sum_{k=0}^{n+1} a_k x^k = \sum_{k=0}^{n+1} (-1)^{n+1-k} \sigma_{n+1-k}(c_0, c_1, \cdots, c_n) x^k, \tag{2.2}$$

where σ_{n+1-k} denotes the kth elementary symmetric function of $c_0, c_1, \cdots, c_n$, i.e., $\sigma_{n+1-k}(c_0, c_1, \cdots, c_n)$ is the sum of all distinct products of k of the arguments $c_0, c_1, \cdots, c_n$ [12, 26]. The inverse of the Vandermonde matrix can be given by the matrix whose elements are

$$\left(V^{-1}\right)_{ij} = \frac{(-1)^{n-j} \sigma_{n-j}(c_0, c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n)}{\prod\limits_{l=0, l \ne i}^{n} (c_i - c_l)}. \tag{2.3}$$

Now we introduce the following lemma before giving the explicit formula for $\boldsymbol{W}$.

Lemma 2.1. *Let σ_{n-k} be the kth elementary symmetric function of $c_0, c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n$, we have*

$$\sum_{k=0}^{n} (-1)^{n-k} k c_j^{k-1} \sigma_{n-k}(c_0, c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n) = \begin{cases} \prod\limits_{\substack{k=0 \\ k \ne i,j}}^{n} (c_j - c_k), & i \ne j, \\ \sum\limits_{\substack{m=0 \\ m \ne i}}^{n} \prod\limits_{\substack{k=0 \\ k \ne i,m}}^{n} (c_j - c_k), & i = j. \end{cases}$$

Proof. Define $\psi(x) = \prod\limits_{k=0, k \ne i}^{n} (x - c_k)$, the derivative of $\psi(x)$

$$\psi'(x) = \sum_{\substack{m=0 \\ m \ne i}}^{n} \prod_{\substack{k=0 \\ k \ne i,m}}^{n} (x - c_k). \tag{2.4}$$

Similar to (2.2), we obtain

$$\psi(x) = \sum_{k=0}^{n} b_k x^k = \sum_{k=0}^{n} (-1)^{n-k} \sigma_{n-k}(c_0, c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n) x^k. \tag{2.5}$$

From (2.5), we see that the derivative of $\psi(x)$ is

$$\psi'(x)=\sum_{k=0}^{n}(-1)^{n-k}k\sigma_{n-k}(c_0,c_1,\cdots,c_{i-1},c_{i+1},\cdots,c_n)x^{k-1}. \tag{2.6}$$

Combining (2.4) and (2.6) implies

$$\sum_{k=0}^{n}(-1)^{n-k}k\sigma_{n-k}(c_0,c_1,\cdots,c_{i-1},c_{i+1},\cdots,c_n)x^{k-1}=\sum_{\substack{m=0\\ m\neq i}}^{n}\prod_{\substack{k=0\\ k\neq i,m}}^{n}(x-c_k). \tag{2.7}$$

Setting $x=c_j$ in (2.7), we have

$$\begin{aligned}&\sum_{k=0}^{n}(-1)^{n-k}kc_j^{k-1}\sigma_{n-k}(c_0,c_1,\cdots,c_{i-1},c_{i+1},\cdots,c_n)\\ &=\sum_{\substack{m=0\\ m\neq i}}^{n}\prod_{\substack{k=0\\ k\neq i,m}}^{n}(c_j-c_k)=\begin{cases}\prod\limits_{\substack{k=0\\ k\neq i,j}}^{n}(c_j-c_k), & i\neq j,\\ \sum\limits_{\substack{m=0\\ m\neq i}}^{n}\prod\limits_{\substack{k=0\\ k\neq i,m}}^{n}(c_i-c_k), & i=j.\end{cases}\end{aligned}$$

Which completes the proof. □

Define the generalized binomial coefficient $\binom{c_n}{c_i}$ $(0<i<n)$ such that

$$\begin{aligned}\binom{c_n}{c_i}&=\frac{(c_n-c_0)(c_n-c_1)\cdots(c_n-c_{n-1})}{[(c_i-c_0)(c_i-c_1)\cdots(c_i-c_{i-1})][(c_n-c_i)(c_{n-1}-c_i)\cdots(c_{i+1}-c_i)]}\\ &=\frac{\prod\limits_{k=0}^{n-1}(c_n-c_k)}{\prod\limits_{k=0}^{i-1}(c_i-c_k)\prod\limits_{k=0}^{n-i-1}(c_{n-k}-c_i)}\\ &=\frac{\prod\limits_{k=0}^{n-1}(c_n-c_k)}{(-1)^{n-i}\prod\limits_{k=0,k\neq i}^{n}(c_i-c_k)}.\end{aligned} \tag{2.8}$$

Further, define

$$\binom{c_n}{c_n}=1\quad\text{and}\quad\binom{c_n}{c_0}=\frac{(-1)^n\prod\limits_{k=0}^{n-1}(c_n-c_k)}{\prod\limits_{k=0}^{n-1}(c_{n-k}-c_0)}. \tag{2.9}$$

Setting $c_n=n$ and $c_i=i$, it is clear that $\binom{c_n}{c_i}$ is the binomial coefficient

$$\binom{n}{i}=\frac{n!}{i!(n-i)!},$$

where $n!$ denotes the factorial of n.

Next we present the main theorem of this paper.

Theorem 2.2. *Let c_i $(0 \le i \le n)$ be distinct real or complex numbers, the elements of $\boldsymbol{W}$ in (1.1) are given by*

$$\boldsymbol{W}_{ij} = \begin{cases} \dfrac{(-1)^{j-i}}{c_j - c_i} \dfrac{\dbinom{c_n}{c_i}}{\dbinom{c_n}{c_j}}, & i \ne j, \\ \displaystyle\sum_{k=0, k\ne i}^{n} \dfrac{1}{(c_i - c_k)}, & i = j. \end{cases} \tag{2.10}$$

Proof. From the definition of $\boldsymbol{H}$ and $\boldsymbol{V}$ in (1.1), it is clear that the elements of $\boldsymbol{HV}$ are

$$(\boldsymbol{HV})_{ij} = i c_j^{i-1}. \tag{2.11}$$

It follows from (1.1), (2.3), (2.11) and Lemma 2.1 that

$$\boldsymbol{W}_{ij} = \sum_{k=0}^{n} \left(V^{-1}\right)_{ik} (\boldsymbol{HV})_{kj} = \sum_{k=0}^{n} \frac{(-1)^{n-k} k c_j^{k-1} \sigma_{n-k}(c_0, c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n)}{\prod\limits_{l=0, l\ne i}^{n} (c_i - c_l)}$$

$$= \begin{cases} \dfrac{\prod\limits_{\substack{k=0\\k\ne i,j}}^{n} (c_j - c_k)}{\prod\limits_{\substack{l=0\\l\ne i}}^{n} (c_i - c_l)}, & i \ne j, \\ \dfrac{\sum\limits_{\substack{m=0\\m\ne i}}^{n} \prod\limits_{\substack{k=0\\k\ne i,m}}^{n} (c_j - c_k)}{\prod\limits_{l=0, l\ne i}^{n} (c_i - c_l)}, & i = j, \end{cases} = \begin{cases} \dfrac{1}{c_j - c_i} \dfrac{\prod\limits_{k=0, k\ne j}^{n} (c_j - c_k)}{\prod\limits_{l=0, l\ne i}^{n} (c_i - c_l)}, & i \ne j, \\ \displaystyle\sum_{m=0, m\ne i}^{n} \dfrac{1}{c_j - c_m}, & i = j. \end{cases} \tag{2.12}$$

Combining (2.8), (2.9) and (2.12) implies

$$\boldsymbol{W}_{ij} = \begin{cases} \dfrac{1}{c_j - c_i} \dfrac{\prod\limits_{k=0, k\ne j}^{n} (c_j - c_k)}{\prod\limits_{k=0, k\ne i}^{n} (c_i - c_k)}, & i \ne j, \\ \displaystyle\sum_{k=0, k\ne i}^{n} \dfrac{1}{c_i - c_k}, & i = j, \end{cases} = \begin{cases} \dfrac{(-1)^{j-i}}{c_j - c_i} \dfrac{\dbinom{c_n}{c_i}}{\dbinom{c_n}{c_j}}, & i \ne j, \\ \displaystyle\sum_{k=0, k\ne i}^{n} \dfrac{1}{(c_i - c_k)}, & i = j. \end{cases} \tag{2.13}$$

Which completes the proof. ☐

Corollary 2.3. *Let $c_i = i$ $(0 \le i \le n)$, the elements of the matrix $\boldsymbol{W}$ in (1.1) are*

$$\boldsymbol{W}_{ij} = \begin{cases} \displaystyle\sum_{p=1}^{i} \frac{1}{p} - \sum_{p=1}^{n-i} \frac{1}{p}, & i = j, \\ \dfrac{(-1)^{j-i}}{j - i} \dfrac{\dbinom{n}{i}}{\dbinom{n}{j}}, & i \ne j. \quad i, j = 0, 1, \ldots, n. \end{cases}$$

Remark 1. *For the case $i = j$, setting $p = i - k$, we have*

$$\sum_{k=0,k\neq i}^{n} \frac{1}{i-k} = \sum_{k=0}^{i-1} \frac{1}{i-k} + \sum_{k=i+1}^{n} \frac{1}{i-k} = \sum_{p=1}^{i} \frac{1}{p} - \sum_{p=1}^{n-i} \frac{1}{p}.$$

Corollary 2.3 was presented in [2]. However, the techniques used in the proof can not be extended for the proof of Theorem 2.2.

2.2. The explicit formula for the inverse of IRK matrix and BIM matrix. The IRK methods based on Gaussian and Radau quadrature formulae possess favorable stability properties, offer a high order of accuracy, and have no start-up issues. In this section, we present the explicit formula for the coefficient matrix of Gauss and Radau IIA methods. The elements of these matrices are given by

$$a_{ij} = \int_0^{c_i} l_j(\theta) d\theta, \tag{2.14}$$

where $l_i(\theta)$ is the Lagrange polynomial $\prod_{k\neq i}(\theta - c_k)/(c_i - c_k)$ and c_i $(1 \leq i \leq n)$ are the zeros of

$$\frac{d^n}{dx^n}\left(x^n(x-1)^n\right), \quad \text{and} \quad \frac{d^n}{dx^{n-1}}\left(x^{n-1}(x-1)^n\right),$$

respectively. Due to $\theta^{k-1} = \sum_{j=1}^{n} c_j^{k-1} l_j(\theta)$ for $k = 1, 2, \ldots, n$, (2.14) is equivalent to the following systems [11]

$$\sum_{j=1}^{n} a_{ij} c_j^{k-1} = \frac{c_i^k}{k}, \quad k = 1, 2, \ldots, n, \quad i = 1, 2, \ldots, n. \tag{2.15}$$

From (2.15), we have

$$\boldsymbol{A} = \boldsymbol{C}_n \boldsymbol{V}_n \boldsymbol{D}_n^{-1} \boldsymbol{V}_n^{-1}, \tag{2.16}$$

where

$$\boldsymbol{C}_n = \begin{pmatrix} c_1 & & & \\ & c_2 & & \\ & & \ddots & \\ & & & c_n \end{pmatrix}, \ \boldsymbol{D}_n = \begin{pmatrix} 1 & & & \\ & 2 & & \\ & & \ddots & \\ & & & n \end{pmatrix}, \ \boldsymbol{V}_n = \begin{pmatrix} 1 & c_1 & \cdots & c_1^{n-1} \\ 1 & c_2 & \cdots & c_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & c_n & \cdots & c_s^{n-1} \end{pmatrix}.$$

Define $\boldsymbol{e}_1 = (1, 0, \cdots, 0)^T \in \mathbb{R}^n$, $\boldsymbol{e} = (1, 1, \cdots, 1)^T \in \mathbb{R}^n$ and $\boldsymbol{c}_0 = (c_0, c_0^2, \cdots, c_0^n)^T$, we have the following theorem.

Theorem 2.4. *Let $c_0 = 0$, $c_i \neq 0$ $(1 \leq i \leq n)$ be distinct real numbers, we have*

$$(\boldsymbol{e}_1^T \boldsymbol{V}_n^{-1} \boldsymbol{C}_n^{-1})_i = \frac{(-1)^i}{-c_i} \frac{\binom{c_n}{c_i}}{\binom{c_n}{c_0}}, \ \left(-\boldsymbol{A}^{-1}\boldsymbol{e}\right)_i = \frac{(-1)^i}{c_i} \frac{\binom{c_n}{c_0}}{\binom{c_n}{c_i}} \ \textit{and}$$

$$\left(\boldsymbol{A}^{-1}\right)_{ij} = \begin{cases} \dfrac{(-1)^{i-j}}{c_i - c_j} \dfrac{\binom{c_n}{c_j}}{\binom{c_n}{c_i}}, & i \neq j, \\ \displaystyle\sum_{k=0,k\neq i}^{n} \frac{1}{c_i - c_k}, & i = j. \end{cases}$$

Proof. It is clear that $\boldsymbol{c}_0 = (0, 0, \cdots, 0)^T \in \mathbb{R}^n$ since $c_0 = 0$. From the definition of the Vandermonde matrix $\boldsymbol{V}$, we have

$$\boldsymbol{V}^{-T} = \begin{pmatrix} 1 & \boldsymbol{c}_0^T \\ \boldsymbol{e} & \boldsymbol{C}_n \boldsymbol{V}_n \end{pmatrix}^{-1} = \begin{pmatrix} 1 & \boldsymbol{c}_0^T \\ -\boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1}\boldsymbol{e} & \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1} \end{pmatrix}, \tag{2.17}$$

Set

$$\boldsymbol{F} = \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ & & & 0 \end{pmatrix}_{(n+1)\times(n+1)}.$$

A straightforward computation implies

$$\boldsymbol{V}^T \boldsymbol{F} = \begin{pmatrix} 0 & \boldsymbol{e}_1^T \\ \boldsymbol{c}_0 & \boldsymbol{V}_n \end{pmatrix} \quad \text{and} \quad \boldsymbol{H}^T = \boldsymbol{F} \begin{pmatrix} 0 & \boldsymbol{c}_0^T \\ \boldsymbol{c}_0 & \boldsymbol{C}_n \end{pmatrix}. \tag{2.18}$$

It follows from (2.17) and (2.18) that

$$\begin{aligned} \boldsymbol{W}^T = \boldsymbol{V}^T \boldsymbol{H}^T \boldsymbol{V}^{-T} &= \boldsymbol{V}^T \boldsymbol{F} \begin{pmatrix} 0 & \boldsymbol{c}_0^T \\ \boldsymbol{c}_0 & \boldsymbol{D}_n \end{pmatrix} \begin{pmatrix} 1 & \boldsymbol{c}_0^T \\ -\boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1}\boldsymbol{e} & \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1} \end{pmatrix} \\ &= \begin{pmatrix} 0 & \boldsymbol{e}_1^T \\ \boldsymbol{c}_0 & \boldsymbol{V}_n \end{pmatrix} \begin{pmatrix} 0 & \boldsymbol{c}_0^T \\ \boldsymbol{c}_0 & \boldsymbol{D}_n \end{pmatrix} \begin{pmatrix} 1 & \boldsymbol{c}_0^T \\ -\boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1}\boldsymbol{e} & \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1} \end{pmatrix} \\ &= \begin{pmatrix} 0 & \boldsymbol{e}_1^T \boldsymbol{D}_n \\ \boldsymbol{c}_0 & \boldsymbol{V}_n \boldsymbol{D}_n \end{pmatrix} \begin{pmatrix} 1 & \boldsymbol{c}_0^T \\ -\boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1}\boldsymbol{e} & \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1} \end{pmatrix} \\ &= \begin{pmatrix} -\boldsymbol{e}_1^T \boldsymbol{D}_n \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1}\boldsymbol{e} & \boldsymbol{e}_1^T \boldsymbol{D}_n \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1} \\ -\boldsymbol{V}_n \boldsymbol{D}_n \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1}\boldsymbol{e} & \boldsymbol{V}_n \boldsymbol{D}_n \boldsymbol{V}_n^{-1}\boldsymbol{C}_n^{-1} \end{pmatrix}. \qquad (2.19) \end{aligned}$$

Combining (2.16), (2.19) and Theorem 2.2 completes the proof. ∎

Remark 2. *From Theorem 2.4, we see that the inverse of IRK matrix can be given explicitly by choosing c_i as the zeros of the shifted Legendre polynomial of degree n.*

Recently, A family of A-stable BIM for solving parabolic equations has been introduced in [17]. BIM have the same desirable properties as IRK methods and are time-parallel in the sense that they compute the solutions at several time steps simultaneously. It shows that the inverse of the BIM matrix can be also given explicitly, i.e.,

$$\boldsymbol{N} = \boldsymbol{D}_n(\boldsymbol{V}_n \boldsymbol{D}_n \boldsymbol{V}_n^{-1} \boldsymbol{D}_n^{-1}) \boldsymbol{D}_n^{-1} + \boldsymbol{D}_n^{-1},$$

where $\boldsymbol{D}_n$ and $\boldsymbol{V}_n$ are defined in (2.16) and $c_i = i$. The explicit expression of $\boldsymbol{N}$ plays an important role in constructing new BIM with positive definite BIM matrices. Then the traditional finite element theory for parabolic equations discretized by the backward Euler or Crank-Nicolson schemes can also be extended to BIM.

3. The fast algorithms for the inverse of the matrix. The explicit formula for $\boldsymbol{W}$ is suitable for symbolic operations using a symbolic manipulation package. However, it may be very expensive for large n. In this section, we present three fast numerical algorithms with $O(n^2)$ complexity to compute $\boldsymbol{W}$.

Following the idea of Traub'algorithm for the inverse of Vandermonde matrix $\boldsymbol{V}$ [26], we introduce the first algorithm to compute the elements of $\boldsymbol{W}$. Firstly, we calculate the coefficients a_k of the monic polynomial $\varphi(x)$ in (2.2) recursively by

$$\begin{cases} \beta_{m+1,m+1} = 1, \quad \beta_{m+1,i} = \beta_{m,i-1} - c_m \beta_{m,i}, & 1 \le m \le n,\ 0 \le i \le m, \\ a_k = \beta_{n+1,k}, & 0 \le k \le n+1, \end{cases}$$

where $\beta_{1,0} = -c_0$, $\beta_{1,1} = 1$ and $\beta_{m,-1} = 0$. Then, we calculate the coefficients b_k of $\psi(x)$ in (2.5) by

$$b_k = a_{k+1} + c_i b_{k+1}, \quad n \ge k \ge 0,$$

where $b_{n+1} = 0$. Finally, we calculate the scalars $\psi(c_i)$ by Horner'rule and obtain

$$\boldsymbol{W}_{ij} = \begin{cases} \dfrac{1}{c_j - c_i} \dfrac{\psi(c_j)}{\psi(c_i)}, & i \ne j, \\ \displaystyle\sum_{k=0, k\ne i}^{n} \frac{1}{c_i - c_k}, & i = j, \end{cases} \tag{3.1}$$

The above scheme is implemented thoroughly in Algorithm 3.1. We see that the computational complexity of Algorithm 3.1 is $11n^2 + 12n$ flops. However, it may soon exceed the representable machine number and produce overflow (or underflow) when computing the coefficients a_k and b_k for large n, this issue also occurs in computing $\psi(c_i)$ since it involving a product of n factors.

In order to avoid the above issue and improve accuracy, we present another fast algorithm by using the special structure of $\boldsymbol{W}$. Firstly, we introduce the matrix $\boldsymbol{G}$ which is given by

$$\boldsymbol{G} = \begin{pmatrix} 1 & c_0 - c_1 & c_0 - c_2 & \cdots & c_0 - c_{n-1} & c_0 - c_n \\ c_1 - c_0 & 1 & c_1 - c_2 & \cdots & c_1 - c_{n-1} & c_1 - c_n \\ c_2 - c_0 & c_2 - c_1 & 1 & \cdots & c_2 - c_{n-1} & c_2 - c_n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n-1} - c_0 & c_{n-1} - c_1 & c_{n-1} - c_2 & \cdots & 1 & c_{n-1} - c_n \\ c_n - c_0 & c_n - c_1 & c_n - c_2 & \cdots & c_n - c_{n-1} & 1 \end{pmatrix}. \tag{3.2}$$

It is clear the computational complexity for computing $\boldsymbol{G}$ is $(n+1)^2$. From the definition of $\psi(x)$, we have

$$\psi(c_i) = \prod_{k=0, k\ne i}^{n} (c_i - c_k) = \prod_{k=0}^{n} \boldsymbol{G}_{ik}. \tag{3.3}$$

Define

$$\widetilde{\boldsymbol{W}} = \begin{pmatrix} 1 & \dfrac{\psi(c_1)}{\psi(c_0)} & \dfrac{\psi(c_2)}{\psi(c_0)} & \cdots & \dfrac{\psi(c_{n-1})}{\psi(c_0)} & \dfrac{\psi(c_n)}{\psi(c_0)} \\ \dfrac{\psi(c_0)}{\psi(c_1)} & 1 & \dfrac{\psi(c_2)}{\psi(c_1)} & \cdots & \dfrac{\psi(c_{n-1})}{\psi(c_1)} & \dfrac{\psi(c_n)}{\psi(c_1)} \\ \dfrac{\psi(c_0)}{\psi(c_2)} & \dfrac{\psi(c_1)}{\psi(c_2)} & 1 & \cdots & \dfrac{\psi(c_{n-1})}{\psi(c_2)} & \dfrac{\psi(c_n)}{\psi(c_2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \dfrac{\psi(c_0)}{\psi(c_{n-1})} & \dfrac{\psi(c_1)}{\psi(c_{n-1})} & \dfrac{\psi(c_2)}{\psi(c_{b-1})} & \cdots & 1 & \dfrac{\psi(c_n)}{\psi(c_{b-1})} \\ \dfrac{\psi(c_0)}{\psi(c_n)} & \dfrac{\psi(c_1)}{\psi(c_n)} & \dfrac{\psi(c_2)}{\psi(c_n)} & \cdots & \dfrac{\psi(c_{n-1})}{\psi(c_n)} & 1 \end{pmatrix}. \tag{3.4}$$

Algorithm 3.1 Given $c(0) < c(1) < \cdots < c(n)$ this algorithm computes $\boldsymbol{W} = \boldsymbol{V}^{-1}\boldsymbol{H}\boldsymbol{V}$

```
procedure EUCLID( $c(i), i = 0, 1, \cdots, n$)
    % Step 1: Computing $a_k$ with $n^2 + 2n$ flops
    $a(0) = -c(0); a(1) = 1$
    for $m = 1; m < n + 1; k + +$ do
        $a(m + 1) = 1$
        for $k = m; k > 0; k - -$ do
            $a(k) = a(k - 1) - c(m) * a(k)$
        end for
        $a(0) = -c(m) * a(0)$
    end for
    % Step 2: Computing $b_k$ and $\psi(c_i) = s(i)$ with $4n^2 + 4n$ flops
    for $i = 0; i < n + 1; i + +$ do
        $b(n) = 1; s(i) = 1$
        for $k = n - 1; k >= 0; k - -$ do
            $b(k) = a(k + 1) + c(i) * b(k + 1)$
            $s(i) = c(i) * s(i) + b(k)$
        end for
    end for
    % Step 3: Computing $W_ij$ with $6n^2 + 6n$ flops
    for $i = 0; i < n + 1; i + +$ do
        $W(i, i) = 0$
        for $j = 0; j < n + 1; j + +$ do
            if $j! = i$ then
                $W(i, j) = s(j)/[(c(j) - c(i)) * s(i)]$
                $W(i, i) = W(i, i) + 1/(c(i) - c(j))$
            end if
        end for
    end for
end procedure
```

From (3.1) and (3.4), we have

$$\boldsymbol{W}_{ij} = \begin{cases} \dfrac{1}{c_j - c_i}\widetilde{\boldsymbol{W}}_{ij}, & i \neq j, \\ \sum\limits_{k=0,k\neq i}^{n} \dfrac{1}{\boldsymbol{G}_{ik}}, & i = j. \end{cases} \tag{3.5}$$

If $\widetilde{\boldsymbol{W}}_{ij}$ have been derived, the complexity for computing $\boldsymbol{W}$ is $3n^2 + 7n$ by using formula (3.5). Next we mainly study how to compute the elements of $\widetilde{\boldsymbol{W}}$.

We first compute the first row of $\widetilde{\boldsymbol{W}}$ by

$$\widetilde{\boldsymbol{W}}_{0j} = \frac{\psi(c_j)}{\psi(c_0)} = \frac{\prod\limits_{k=0}^{n} \boldsymbol{G}_{jk}}{\prod\limits_{k=0}^{n} \boldsymbol{G}_{0k}} = \prod_{k=0}^{n} \frac{\boldsymbol{G}_{jk}}{\boldsymbol{G}_{0k}}, \quad 0 \leq j \leq n. \tag{3.6}$$

Instead of computing the numerator $\psi(c_j)$ and the denominator $\psi(c_0)$ for each j, we compute $\widetilde{\boldsymbol{W}}_{0j}$ by a product of $n+1$ factors $\boldsymbol{G}_{jk}/\boldsymbol{G}_{0k}$. This trick may avoid overflow

Algorithm 3.2 Given $c(0) < c(1) < \cdots < c(n)$ this algorithm computes $\boldsymbol{W} = \boldsymbol{V}^{-1}\boldsymbol{H}\boldsymbol{V}$

```
procedure EUCLID( $c(i), i = 0, 1, \cdots, n$ )
    % Step 1: Computing $\boldsymbol{G}_{ij}$ with $n^2 + 3n$ flops
    for $i = 0; i < n + 1; i++$ do
        for $j = 0; j < n + 1; j++$ do
            if $j! = i$ then
                $G(i, j) = c(i) - c(j)$
            end if
            $G(i, i) = 1$
        end for
    end for
    % Step 2a: Computing the first row of $\widetilde{\boldsymbol{W}}$ and g with $2n^2 + 5n$ flops
    for $j = 0; j < n + 1; j++$ do
        for $k = 0; k < n + 1; k++$ do
            $W(0, j) = W(0, j) * (G(j, k)/G(0, k))$
        end for
        $g(j) = 1/W(0, j)$
    end for
    % Step 2b: Computing $\widetilde{\boldsymbol{W}}_{ij}$ with $n^2 + n$ flops
    for $i = 1; i < n + 1; i++$ do
        for $j = 0; j < n + 1; j++$ do
            $W(i, j) = W(0, j) * g(i)$
        end for
    end for
    % Step 3: Computing $\boldsymbol{W}_{ij}$ with $2n^2 + 5n$ flops
    for $i = 0; i < n + 1; i++$ do
        for $j = 0; j < n + 1; j++$ do
            $W(i, i) = W(i, i) + (1/G(i, j))$
            $W(i, j) = W(i, j)/G(j, i)$
        end for
        $W(i, i) = W(i, i) - 2$
    end for
end procedure
```

or underflow and improve accuracy. It is clear that the complexity for computing $\widetilde{\boldsymbol{W}}_{0j}$ $(0 \le j \le n)$ is $2n^2 + 4n$. Set

$$\mathbf{g} = \left(\frac{1}{\widetilde{\boldsymbol{W}}_{01}}, \frac{1}{\widetilde{\boldsymbol{W}}_{02}}, \cdots, \frac{1}{\widetilde{\boldsymbol{W}}_{0n}}\right)^T, \tag{3.7}$$

the elements $\widetilde{\boldsymbol{W}}$ can be compute with $n^2 + n$ operations by

$$\widetilde{\boldsymbol{W}}_{ij} = \frac{\psi(c_j)}{\psi(c_i)} = \frac{\psi(c_j)}{\psi(c_0)}\frac{\psi(c_0)}{\psi(c_i)} = \widetilde{\boldsymbol{W}}_{0j}\boldsymbol{g}_i, \quad 0 \le j \le n,\ 1 \le i \le n. \tag{3.8}$$

From the above analysis, we observe that it requires $7n^2 + 15n$ operations to compute $\boldsymbol{W}$. The details of this scheme is presented in Algorithm 3.2.

From (3.6) and (3.7), we see that the overflow and underflow may also occur if

Algorithm 3.3 Given $c(0) < c(1) < \cdots < c(n)$ this algorithm computes $\boldsymbol{W} = \boldsymbol{V}^{-1}\boldsymbol{H}\boldsymbol{V}$

```
procedure EUCLID( $c(i), i = 0, 1, \cdots, n$)
    % Step 1: Computing $\boldsymbol{G}_{ij}$ with $n^2 + n$ flops
    for $i = 0; i < n + 1; i++$ do
        for $j = 0; j < n + 1; j++$ do
            if $j! = i$ then
                $G(i, j) = c(i) - c(j)$
            end if
            $G(i, i) = 1$
        end for
    end for
    % Step 2a: Computing $\widetilde{\boldsymbol{g}}$ with $2n^2 - 2$ flops
    for $i = 0; i < n; i++$ do
        $g(i) = -1$
        for $k = 0; k < n + 1; k++$ do
            if $(k! = i)\&\&(k! = i + 1)$ then
                $g(i) = g(i) * (G(i + 1, k)/G(i, k))$
            end if
        end for
    end for
    % Step 2b: Computing $\widetilde{\boldsymbol{W}}_{ij}$ with $n^2 + n$ flops
    for $i = 0; i < n + 1; i++$ do
        $W(i, i) = 1$
        for $j = i + 1; j < n + 1; j++$ do
            $W(i, j) = W(i, j - 1) * g(j - 1)$
            $W(j, i) = 1/W(i, j)$
        end for
    end for
    % Step 3: Computing $\boldsymbol{W}_{ij}$ with $3n^2 + 7n$ flops
    for $i = 0; i < n + 1; i++$ do
        for $j = 0; j < n + 1; j++$ do
            $W(i, i) = W(i, i) + (1/G(i, j))$
            $W(i, j) = W(i, j)/G(j, i)$
        end for
        $W(i, i) = W(i, i) - 2$
    end for
end procedure
```

$\boldsymbol{G}_{jk}$ is far greater or less than $\boldsymbol{G}_{0k}$. Now we introduce vector $\widetilde{\boldsymbol{g}}$ which is given by

$$\widetilde{\mathbf{g}} = \left(\frac{\psi(c_1)}{\psi(c_0)}, \frac{\psi(c_2)}{\psi(c_1)}, \cdots, \frac{\psi(c_n)}{\psi(c_{n-1})}\right)^T, \tag{3.9}$$

where $\widetilde{\boldsymbol{g}}_i$ $(0 \le i \le n-1)$ are computed by

$$\begin{aligned}
\widetilde{\boldsymbol{g}}_i &= \frac{\psi(c_{i+1})}{\psi(c_i)} \\
&= \frac{(c_{i+1}-c_0)(c_{i+1}-c_1)\cdots(c_{i+1}-c_{i-1})(c_{i+1}-c_i)(c_{i+1}-c_{i+2})\cdots(c_{i+1}-c_n)}{(c_i-c_0)(c_i-c_1)\cdots(c_i-c_{i-1})(c_i-c_{i+1})(c_i-c_{i+2})\cdots(c_i-c_n)} \\
&= -\frac{(c_{i+1}-c_0)(c_{i+1}-c_1)\cdots(c_{i+1}-c_{i-1})(c_{i+1}-c_{i+2})\cdots(c_{i+1}-c_n)}{(c_i-c_0)(c_i-c_1)\cdots(c_i-c_{i-1})(c_i-c_{i+2})\cdots(c_i-c_n)} \\
&= -\prod_{\substack{k=0 \\ k\neq i,i+1}}^{n} \frac{(c_{i+1}-c_k)}{(c_i-c_k)} = -\prod_{\substack{k=0 \\ k\neq i,i+1}}^{n} \frac{G_{(i+1)k}}{G_{ik}}.
\end{aligned} \tag{3.10}$$

If $c_0 < c_1 < \cdots < c_n$, it is clear that (3.10) may be more accurate than (3.6). From (3.4), we observe that the upper triangular part of $\widetilde{\boldsymbol{W}}$ can be computed recursively by

$$\widetilde{\boldsymbol{W}}_{ij} = \frac{\psi(c_j)}{\psi(c_i)} = \frac{\psi(c_{j-1})}{\psi(c_i)}\frac{\psi(c_j)}{\psi(c_{j-1})} = \widetilde{\boldsymbol{W}}_{i(j-1)}\widetilde{\boldsymbol{g}}_{j-1}, \quad i+1 \le j \le n,\ 0 \le i \le n, \tag{3.11}$$

where $\widetilde{\boldsymbol{W}}_{ii} = 1$. Then the strictly lower triangular part of $\widetilde{\boldsymbol{W}}$ is given by

$$\widetilde{\boldsymbol{W}}_{ij} = \frac{1}{\widetilde{\boldsymbol{W}}_{ji}}, \quad 0 \le j \le i-1,\ 1 \le i \le n, \tag{3.12}$$

A straightforward calculation shows that complexity for computing $\widetilde{\boldsymbol{g}}$ and $\widetilde{\boldsymbol{W}}$ are $2n^2$ and n^2+n, respectively. Finally, we obtain Algorithm 3.3 with the complexity $7n^2+9n$ flops.

Both Algorithm 3.2 and Algorithm 3.3 include three steps, i.e., computing the elements of the matrices $\boldsymbol{G}$, $\widetilde{\boldsymbol{W}}$ and $\boldsymbol{W}$ sequentially. The difference between these algorithms lies in the computation of $\widetilde{\boldsymbol{W}}$ in the second step. In each step, both Algorithm 3.2 and Algorithm 3.3 can be implemented in parallel since the computations of each row of $\boldsymbol{G}$, $\widetilde{\boldsymbol{W}}$ and $\boldsymbol{W}$ are independent.

4. Numerical experiments.

In this section, we present some symbolic and numerical results to demonstrate the performance of the proposed formula and algorithms. In the experiments, we calculate $\boldsymbol{W}$ by choosing c_i $(i = 1, 2, \cdots, n)$ from the folowing four examples.

Example 1. *(Gauss nodes in (0,1)): $c_0 = 0$ and c_i $(i = 1, 2, \cdots, n)$ are the zeros of $\frac{d^n}{dx^n}\left(x^n(x-1)^n\right)$.*

Example 2. *(Radau IIA nodes in (0,1]): $c_0 = 0$ and c_i $(i = 1, 2, \cdots, n)$ are the zeros of $\frac{d^n}{dx^{n-1}}\left(x^{n-1}(x-1)^n\right)$.*

Example 3. *(Chebyshev nodes in (0,1)): $c_i = \frac{1}{2}+\frac{1}{2}\cos\left(\frac{2i+1}{2n+1}\pi\right)$ $(i = 0, 1, 2, \cdots, n)$.*

Example 4. *(Equidistant nodes in [0,n]): $c_i = i$ $(i = 0, 1, 2, \cdots, n)$.*

4.1. Results of the symbolic operations.

Since $\boldsymbol{W}$ is given explcitly by (3.1), we can compute it exactly by Mathematica. From Theorem 2.4, we see that the submatrix $\boldsymbol{W}_n^T$ is exactly the inverse of the IRK matrix $\boldsymbol{A}$ when choosing c_i as in Examples 1 and 2. Note that the coefficient matrix of Gauss and Radau IIA methods can be calculated using the Mathematica command *ImplicitRungeKuttaGaussCoefficients* and *ImplicitRungeKuttaRadauIIACoefficients*, which allow software-implemented precision of arbitrary numbers of digits. The number of digits was set to 1000 in the experiment To test the performance of the explicit formula (3.1), we define

$$Err = \|\boldsymbol{W}_n^T \boldsymbol{A} - \boldsymbol{I}\|_\infty \tag{4.1}$$

and list the results in Tables 1 and 2, respectively. It is clear that our formula (3.1) is correct, and the compute time of $\boldsymbol{A}$ and $\boldsymbol{W}_n$ grows fast with the increase of n. However, the calculation of $\boldsymbol{W}_n$ is much faster that of $\boldsymbol{A}$. The calculation of IRK matrix using Mathematica command takes significantly more time (seconds) compared to computing its inverse using the formula (3.1).

Table 1
Symbolic computations for $\boldsymbol{A}$ and $\boldsymbol{W}_n$ with Gauss nodes in Example 1

n	$\boldsymbol{A}$	$\boldsymbol{W}_n$	Err
5	5.70e-4	7.53e-3	3.62e-71
10	1.44e-3	1.04e-2	5.93e-71
20	7.23e-3	1.60e-2	1.31e-70
50	1.02e-1	5.68e-2	3.92e-70
100	9.21e-1	2.13e-1	6.11e-70
200	9.72e0	1.29e0	1.25e-69
500	2.75e2	5.54e1	2.97e-69
1000	3.93e3	6.51e2	5.65e-69

Table 2
Symbolic computations for $\boldsymbol{A}$ and $\boldsymbol{W}_n$ with Radau IIA nodes in Example 2

n	$\boldsymbol{A}$	$\boldsymbol{W}_n$	Err
5	6.37e-4	8.85e-3	3.62e-71
10	1.41e-3	1.09e-2	6.24e-71
20	7.05e-3	1.49e-2	1.09e-70
50	1.02e-1	5.98e-2	2.75e-70
100	9.16e-1	2.10e-1	5.89e-70
200	9.66e0	1.32e0	1.05e-69
500	2.73e2	5.52e1	2.73e-69
1000	3.89e3	6.81e2	5.39e-69

4.2. Results of the fast algorithms. Symbolic operations are time-consuming, although they allow for arbitrary precision. Now we will test the performance of Algorithms 3.2 and 3.3 on Examples 1-4. In the experiments, the matrices computed by using long double (80 bits) and quadruple precision (128 bits) are denoted by $\boldsymbol{W}^{\boldsymbol{d}}$ and $\boldsymbol{W}^q$, respectively. The compute time is denoted by "T_d"and "T_q". We consider $\boldsymbol{W}^q$ as exact and define the infinite–norm and componentwise relative errors as follows:

$$Err_1 = \frac{\|\boldsymbol{W}^d - \boldsymbol{W}^q\|_\infty}{\|\boldsymbol{W}^q\|_\infty} \quad \text{and} \quad Err_2 = \max_{0\le i,j\le n}\left\{\frac{|\boldsymbol{W}^d_{ij} - \boldsymbol{W}^q_{ij}|}{|\boldsymbol{W}^q_{ij}|}\right\}. \tag{4.2}$$

The numrical results for Example 1 are presented in Table 3, which shows that both Algorithms 3.2 and 3.3 are very fast although the compute time increases as n

Table 3

Example 1: the CPU time (Seconds) and errors for $\boldsymbol{W}$ computed by two types of precision

	Algorithm 3.2			Algorithm 3.3		
n	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q
5	1.62e-19	2.36e-19/2.45e-19	7.0e-6/4.1e-5	1.20e-19	3.00e-19/2.45e-19	6.0e-6/4.0e-5
10	1.35e-19	7.41e-19/1.89e-18	1.5e-5/1.1e-4	2.37e-19	1.14e-18/1.89e-18	1.6e-5/1.1e-4
20	5.23e-19	3.40e-18/5.01e-18	4.6e-5/3.6e-4	5.75e-19	3.65e-18/5.01e-18	5.0e-5/3.6e-4
50	3.17e-18	1.54e-17/5.05e-17	2.6e-4/1.9e-3	3.73e-18	1.62e-17/5.05e-17	2.7e-4/2.0e-3
100	5.17e-18	4.92e-17/2.17e-16	1.0e-3/7.3e-3	5.98e-18	4.92e-17/2.17e-16	1.0e-3/7.5e-3
200	7.24e-17	6.74e-16/1.05e-15	4.0e-3/2.2e-2	7.20e-17	6.73e-16/1.05e-15	4.0e-3/2.2e-2
500	2.99e-16	1.43e-15/6.65e-10	1.1e-2/5.8e-2	2.99e-16	1.43e-15/6.65e-15	1.1e-2/6.0e-2
1000	2.44e-15	1.35e-14/2.86e-14	2.9e-2/2.1e-1	2.43e-15	1.35e-14/2.86e-14	3.0e-2/2.1e-1
2000	1.06e-15	1.08e-14/1.06e-13	1.5e-1/9.1e-1	1.06e-15	1.08e-14/1.06e-13	1.5e-1/9.5e-1
3000	8.65e-15	7.10e-14/2.61e-13	3.4e-1/2.1	8.64e-15	7.10e-14/2.61e-13	3.4e-1/2.2
4000	6.90e-15	4.21e-14/4.56e-13	6.1e-1/4.0	6.88e-15	4.21e-14/4.56e-13	6.1e-1/4.3
5000	2.41e-14	1.63e-13/6.90e-13	9.4e-1/6.3	2.41e-14	1.63e-13/6.90e-13	1.0/6.7
8000	4.50e-14	4.03e-13/1.95e-12	3.2/17.6	4.50e-14	4.03e-13/1.95e-12	3.5/18.2
10000	–	–	–	7.16e-14	6.39e-13/2.84e-12	7.3/35.6

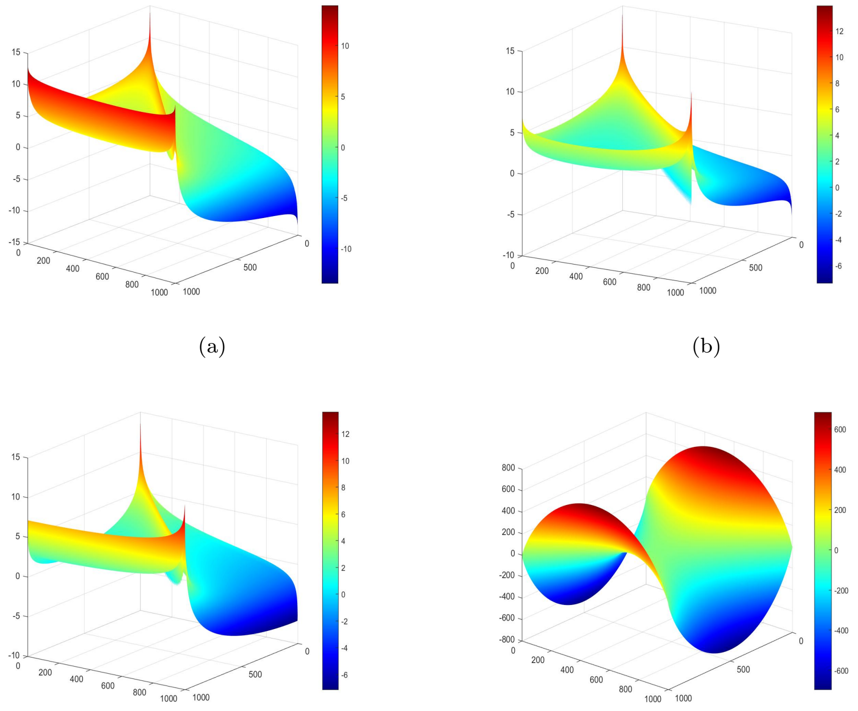
Table 4

Example 2: the CPU time (Seconds) and errors for $\boldsymbol{W}$ computed by two types of precision

	Algorithm 3.2			Algorithm 3.3		
n	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q
5	1.82e-19	3.66e-19/2.17e-18	6.0e-6/4.2e-5	1.23e-19	3.09e-19/2.17e-18	6.0e-6/4.0e-5
10	2.52e-19	1.63e-18/3.90e-17	1.5e-5/1.1e-4	2.23e-19	1.54e-18/3.90e-17	1.6e-5/1.1e-4
20	6.73e-19	5.17e-18/7.41e-16	4.7e-5/3.7e-4	6.84e-19	5.32e-18/7.41e-16	5.0e-5/3.6e-4
50	3.48e-18	2.31e-17/1.26e-14	2.6e-4/1.9e-3	3.41e-18	2.26e-17/1.26e-14	2.7e-4/2.0e-3
100	3.88e-18	3.58e-17/6.64e-14	1.0e-3/7.3e-3	4.15e-18	3.51e-17/6.64e-14	1.0e-3/7.4e-3
200	7.79e-17	5.33e-16/7.20e-12	3.5e-3/2.1e-2	7.77e-17	5.32e-16/7.20e-12	4.0e-3/2.2e-2
500	2.89e-16	2.51e-15/1.35e-10	1.1e-2/6.1e-2	2.89e-16	2.51e-15/1.35e-10	1.1e-2/6.1e-2
1000	1.42e-15	9.61e-15/3.47e-9	3.0e-2/2.5e-1	1.42e-15	9.60e-15/3.47e-9	3.0e-2/2.1e-1
2000	2.06e-15	1.95e-14/1.28e-8	2.0e-1/9.2e-1	2.06e-15	1.95e-14/1.28e-8	1.5e-1/9.6e-1
3000	5.59e-15	3.90e-14/1.01e-7	3.4e-1/2.1	5.59e-15	3.90e-14/1.01e-7	3.4e-1/2.3
4000	1.67e-14	1.39e-13/5.49e-7	5.9e-1/4.0	1.67e-14	1.39e-13/5.49e-7	6.1e-1/4.3
5000	3.83e-14	2.87e-13/2.31e-6	9.5e-1/6.3	3.83e-14	2.87e-13/2.31e-6	1.0/6.4
8000	2.70e-14	1.82e-13/2.86e-6	3.4/16.4	2.70e-14	1.82e-13/2.86e-6	3.5/19.0
10000	–	–	–	1.60e-13	9.81e-13/3.60e-5	7.5/35.6

grows. We observe that the time for computing $\boldsymbol{W}^d$ is no more than one second in the case $n = 5000$. It is clear that the errors in the computation of $\boldsymbol{W}$ grow with n, and the accuracy of off-diagonal elements is slightly better than diagonal elements. From (3.1), we see that the computation of the daagonal elements involving recursive summation vhich may suffers cancellation. Additionally, we observe that Algorithm 3.2 fails for the case $n = 10000$. The reason is that the overflow occurs in the computation of formula (3.6) since G_{jk} is far greater than G_{0k} when j is large. However, Algorithm 3.3 works well since the computation of formula (3.10) can avoid overflow.

Similarly to Example 1, Tables 4 and 5 show that the compute time grows mildly with the increase of n and both Algorithms 3.2 and 3.3 yield very high accuracy in



(a) (b)

(c) (d)

Fig. 1. *3D Heatmap of the matrix elements* ($\ln|\boldsymbol{W}_{ij}|$)*,* $(a), (b), (c), (d)$ *correspond to Examples 1-4* ($n = 1000$) *, respectively*

the infinite–norm. However, the accuracy of the diagonal elements is much worse than the off-diagonal elements if n is large. In the experiments, we observe that there is heavy cancellation in the sum, that is, $\sum\limits_{k=0,k\neq i}^{n} |\frac{1}{c_i - c_k}| \gg \sum\limits_{k=0,k\neq i}^{n} \frac{1}{c_i - c_k}$. Moreover, cancellation also occurs in the subtraction of two nearly equal numbers. Hence, there exists a severe loss of significant digits and produces a large relative error in this particular situation. In Figure 1, we present 3D heatmap of the matrix elements for Examples 1-4 by using $\ln|\boldsymbol{W}_{ij}|$ instead of $\boldsymbol{W}_{ij}$. From the subfigures (a)-(c), we observe that the matrices exhibit a significant difference in element size, with the diagonal elements being small and approaching 1 in subfigure (b), moreover, the term $\frac{1}{c_i - c_k}$ in recursive summation may be very large and is either positive or negative. So the algorithms suffer severe cancellation. Fortunately, we can achieve desired high accuracy for this special example by using sufficiently high precision, such as quadruple precision.

Finally, we test the fourth example and present the numerical results in Table 6, which illustrates that both Algorithms 3.2 and 3.3 are very accurate in both types of errors. From Remark 1, we see that the heavy cancellation does not occur in the sum since there exist no nearly equal numbers are subtracted from each other. Subfigure (d) shows that the values of the matrix entries varies widely, making underflow and overflow unavoidable consequences when working in finite precision arithmetic for very large n.

Table 5

Example 3: the CPU time (Seconds) and errors for $\boldsymbol{W}$ computed by two types of precision

	Algorithm 3.2			Algorithm 3.3		
n	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q
5	2.38e-19	5.55e-19/2.86e-18	7.0e-6/3.8e-5	2.61e-19	6.07e-19/2.86e-18	6.0e-6/3.5e-5
10	6.41e-19	1.70e-18/3.38e-17	1.5e-5/1.1e-4	6.88e-19	1.61e-18/3.38e-17	1.6e-5/1.1e-4
20	3.06e-18	5.85e-18/1.97e-16	4.6e-5/3.7e-4	3.09e-18	5.98e-18/1.97e-16	5.0e-5/3.7e-4
50	1.04e-17	2.25e-17/6.05e-15	2.6e-4/1.9e-3	1.06e-17	2.61e-17/6.05e-15	2.7e-4/2.0e-3
100	1.03e-16	2.31e-16/9.74e-15	1.0e-3/7.3e-3	1.03e-16	2.30e-16/9.74e-15	1.0e-3/7.5e-3
200	1.72e-16	3.78e-16/2.60e-13	3.9e-3/2.0e-2	1.73e-16	3.78e-16/2.60e-13	4.1e-3/2.0e-2
500	2.69e-15	5.86e-15/1.75e-12	1.1e-2/6.0e-2	2.69e-15	5.86e-15/1.75e-12	1.1e-2/6.1e-2
1000	3.22e-15	8.53e-15/3.63e-11	2.7e-2/2.0e-1	3.21e-15	8.53e-15/3.63e-11	3.0e-2/2.4e-1
2000	2.11e-14	5.03e-14/2.02e-10	1.3e-1/9.3e-1	2.11e-14	5.03e-14/2.02e-10	1.5e-1/9.5e-1
3000	1.07e-13	2.45e-13/5.37e-10	3.3e-1/2.1	1.07e-13	2.45e-13/5.37e-10	3.4e-1/2.2
4000	1.01e-13	2.33e-13/1.69e-9	5.9e-1/4.0	1.01e-13	2.33e-13/1.69e-9	6.1e-1/4.1
5000	2.03e-13	4.95e-13/2.44e-9	9.7e-1/6.2	2.03e-13	4.95e-13/2.44e-9	9.7e-1/6.5
8000	5.80e-13	1.39e-12/1.13e-8	3.3/17.3	5.80e-13	1.39e-12/1.13e-8	3.9/17.0
10000	–	–	–	4.12e-13	9.87e-13/4.35e-8	8.1/34.5
15000	–	–	–	1.43e-12	2.70e-12/1.92e-8	15.3/89.4

Table 6

Example 4: the CPU time (Seconds) and errors for $\boldsymbol{W}$ computed by two types of precision

	Algorithm 3.2			Algorithm 3.3		
n	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q	Err_1	$Err_2(i \neq j)/(i = j)$	T_d/T_q
5	7.66e-20	1.20e-19/4.40e-19	7.0e-6/3.8e-5	1.42e-20	5.00e-20/4.40e-19	6.0e-6/3.5e-5
10	9.70e-20	1.72e-19/5.64e-19	1.5e-5/1.1e-4	1.28e-19	3.20e-19/5.64e-19	1.6e-5/1.1e-4
20	8.11e-20	3.68e-19/1.39e-18	4.6e-5/3.2e-4	6.67e-19	1.47e-18/1.39e-18	4.9e-5/3.2e-4
50	2.51e-19	8.83e-19/1.19e-17	2.6e-4/1.7e-3	3.62e-18	7.13e-18/1.19e-17	2.7e-4/1.7e-3
100	7.64e-19	1.68e-18/5.69e-18	1.0e-3/6.4e-3	8.87e-18	1.79e-17/5.68e-18	1.0e-3/6.6e-3
200	7.64e-19	2.25e-18/1.65e-17	4.0e-3/1.9e-2	2.98e-17	5.22e-17/1.65e-17	4.0e-3/2.0e-2
500	1.41e-18	4.05e-18/1.51e-16	1.2e-2/5.7e-2	2.16e-17	5.57e-17/1.51e-16	1.1e-2/5.8e-2
1000	2.68e-18	6.65e-18/3.57e-16	2.8e-2/1.8e-1	1.69e-16	3.58e-16/3.57e-16	3.7e-2/2.5e-1
2000	3.61e-18	1.09e-17/4.44e-16	1.3e-1/8.1e-1	4.97e-16	1.07e-15/4.44e-16	1.6e-1/8.7e-1
3000	2.88e-18	1.48e-17/6.58e-16	3.3e-1/2.0	6.77e-16	1.43e-15/6.58e-16	3.7e-1/2.9
4000	4.65e-18	1.51e-17/1.09e-15	5.8e-1/3.6	1.66e-15	3.36e-15/1.09e-15	6.2e-1/4.7
5000	4.95e-18	1.63e-17/1.04e-15	9.4e-1/5.9	2.34e-15	4.74e-15/1.04e-15	9.9e-1/6.2
8000	7.08e-18	2.42e-17/3.85e-15	3.2/14.9	2.19e-15	5.01e-15/3.85e-15	3.4/18.6
10000	8.12e-18	2.83e-17/2.16e-15	7.2/30.0	3.34e-15	6.69e-15/2.16e-15	9.2/36.4

5. Conclusion. We proposed an explicit formula to compute the inverse of a type of matrices which include the well-known fully IRK and BIM matrices as special cases. So the inverse of these matrices arising from implicit time integration can be computed by the basic four arithmetic operations of the quadrature nodes, and we can obtain their inverse exactly by using a symbolic manipulation package. It's shown that the inverse of these matrices is much easier to compute than themselves by Mathematica. We also presented three fast algorithms with $O(n^2)$ complexity to compute the inverse of thees matrices. The numerical results demonstrate that Algorithm 3.3 is very fast and it only requires several seconds to obtain the inverse of the matrices with size $n = 10000$. Although encountering substantial cancellation in the computation of diagonal elements with Radau IIA nodes and Chebyshev nodes, we can also obtain desirable high accuracy by using higher precision, such as quadruple precision, and the compute time is considerably less than that required by using the symbolic manipulation package.

REFERENCES

[1] O. Axelsson, I., Dravins, and M. Neytcheva, *Stage-parallel preconditioners for implicit Runge–Kutta methods of arbitrarily high order, linear problems*, Numer. Linear Algebra Appl. 31 (2024), e2532.

[2] L. Aceto and D. Trigiante, *On the A-stable methods in the GBDF class*, Nonlinear Anal.: Real World Appl. 3 (2002), pp. 9-23.

[3] T. A. Bickart, *An efficient solution process for implicit Runge–Kutta methods*, SIAM J Nume.r Anal. 14 (1977), pp. 1022–1027.

[4] L. Brugnano and D. Trigiante, *Block implicit methods for ODEs*, Recent trends in numerical analysis, 81-105, Adv. Theory Comput. Math., 3, Nova Sci. Publ., Huntington, NY 2001.

[5] J. C. Butcher, *On the implementation of implicit Runge–Kutta methods*, BIT 16 (1976), pp. 237–240

[6] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*, Third edition. John Wiley & Sons, Ltd., Chichester, 2016.

[7] D. Calvetti amd L. Reichel, *Fast inversion of Vandermonde-like matrices involving orthogonal polynomials*, BIT 33 (1993), pp. 473–484.

[8] I. Gohberg and V. Olshevsky, *Fast inversion of Chebyshev–Vandermonde matrices*, Numer. Math. 67 (1994), pp. 71-92.

[9] I. Gohberg and V. Olshevsky, *The fast generalized Parker-Traub algorithm for inversion of Vandermonde and related matrices*, J. Complex. 13 (1997), pp. 208–234.

[10] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations. II: Stiff and Differential-Algebraic Problems*, Springer, Berlin, 1996.

[11] E. Hairer, C. Lubich, and G. Wanner, *Gerhard Geometric Numerical Integration. Structure-Preserving Algorithms for Ordinary Differential Equations*, Springer, Heidelberg, 2006.

[12] N. J. Higham, em Accuracy and stability of numerical algorithms, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[13] L. O. Jay, *Inexact simplified Newton iterations for implicit Runge-Kutta methods*, SIAM J. Numer. Anal. 38 (2000), pp. 1369-1388.

[14] L. O. Jay and T. Braconnier, *A parallelizable preconditioner for the iterative solution of implicit Runge-Kutta type methods*, J. Comput. Appl. Math. 111 (1999), pp. 63-76.

[15] X. Jiao, X. Wang, and Q. Chen, *Optimal and low-memory near-optimal preconditioning of fully implicit Runge-Kutta schemes for parabolic PDEs*, SIAM J. Sci. Comput. 43 (2021), pp. A3527-A3551.

[16] J. Van lent and S. Vandewalle, *Multigrid methods for implicit Runge-Kutta and boundary value method discretizations of PDEs*, SIAM J. Sci. Comput. 27 (2005), pp. 67-92.

[17] S. Li, J.-Y. Wang, and X.-C. Cai, *A-stable high order block implicit methods for parabolic equations*, SIAM J. Numer. Anal. 61 (2023), pp. 1858-1884.

[18] K.-A. Mardal, T. K. Nilssen, and G. A. Staff, *Order-optimal preconditioners for implicit Runge-Kutta schemes applied to parabolic PDEs*, SIAM J. Sci. Comput. 29 (2007), pp. 361-375.

[19] P. Munch, I. Dravins, M. Kronbichler, and M. Neytcheva, *Stage-parallel fully implicit Runge–Kutta implementations with optimal multilevel preconditioners at the scaling limit*, SIAM J. Sci. Comput. (2023), pp. S71-S96.

[20] W. Pazner and P.-O. Persson, *Stage-parallel fully implicit Runge–Kutta solvers for discontinuous Galerkin fluid simulations*, J. Comput. Phys., 335 (2017), pp. 700-717.

[21] Md. M. Rana, V. E. Howle, K. Long, A. Meek, and W. Milestone, *A new block preconditioner for implicit Runge-Kutta methods for parabolic PDE problems*, SIAM J. Sci. Comput. 43 (2021), pp. S475-S495.

[22] L. F. Shampine and H. A. Watts, *Block implicit one-step methods*, Math. Comp. 23 (1969), pp. 731-740.

[23] M.-R. Skrzipek, *Inversion of vandermonde-like matrices*, BIT, 44 (2004), pp. 291-306

[24] B. S. Southworth, O. Krzysik, W. Pazner, and H. D. Sterck, *Fast Solution of Fully Implicit Runge-Kutta and Discontinuous Galerkin in Time for Numerical PDEs, Part I: the Linear Setting*, SIAM J. Sci. Comput. 44 (2022), pp. A416–A443.

[25] B. S. Southworth, O. Krzysik, and W. Pazner, *Fast solution of fully implicit Runge-Kutta and discontinuous Galerkin in time for numerical PDEs, Part II: Nonlinearities and DAEs*, SIAM J. Sci. Comput. 44 (2022), pp. A636–A663.

[26] J. Traub, *Associated polynomials and uniform methods for the solution of linear problems*, SIAM Rev. 8 (1966), pp. 277–301.

[27] Z. Xu and Z. You, *A fast inversion of confluent Vandermonde-like matrices involving polynomials that satisfy a three-term recurrence relation*, SIAM J. Matrix Anal.Appl. 19 (1998), pp. 797–806.