

1 **KRYLOV SUBSPACE METHOD FOR MATRIX EXPONENTIALS IN
2 THE ACOUSTIC PARABOLIC WAVE EQUATION**

3 WILLIAM R. HARRIS (HARRISWR@MIT.EDU)

4 **Abstract.** In this project, a Krylov subspace method for approximating the forward propagation
5 of an underwater acoustic pressure field through simple underwater environments is implemented in
6 Python. The matrix exponentiation algorithm is validated using a simple 1D heat diffusion equation
7 by comparing to analytic solution. A parabolic equation model is implemented for simple ocean
8 environments. An error analysis for a forward Euler, Crank-Nicholson, and the Krylov subspace
9 matrix exponentiation is completed for a range-independent, isovelocity sound speed profile using a
10 Padé approximation of the full propagator with a small range step as ground truth. It is found that
11 Krylov subspaces allow for larger range steps with small errors in transmission loss. While the Krylov
12 subspace method provides improved computational efficiency compared to directly exponentiating
13 the propagation operators, it does not compare favorably to established Padé methods within the
14 field of ocean acoustics.

15 **Key words.** Krylov subspaces, acoustics, parabolic equation

16 **1. Introduction.** The numerical solution of partial differential equations often
17 requires the computation of matrix exponentials, particularly in problems involving
18 wave propagation, such as underwater acoustics modeled by the Parabolic Equation
19 (PE) method. The matrix exponential, $e^A v$, where A is a large, sparse matrix and
20 v is a vector, arises in the time or range evolution of such systems. Direct computa-
21 tion of e^A for large matrices is computationally prohibitive due to the high cost
22 of dense matrix operations, necessitating efficient approximation techniques. Among
23 these, Krylov subspace methods have emerged as a powerful approach, leveraging the
24 sparsity of the matrix to construct low-dimensional approximations that capture the
25 essential influences of the exponential operator [9]. This introduction outlines the
26 Krylov subspace framework, its computational construction, and its application to
27 matrix exponentiation, setting the stage for its use in the PE method.

28 The Krylov subspace is a fundamental concept in numerical linear algebra, defined
29 as the space spanned by a sequence of vectors generated by repeated application of a
30 matrix A to an initial vector v . Specifically, for a matrix $A \in \mathbb{R}^{n \times n}$ and vector $v \in \mathbb{R}^n$,
31 the Krylov subspace of dimension m is $K_m(A, v) = \text{span}\{v, Av, A^2v, \dots, A^{m-1}v\}$ [4].
32 The key idea is to approximate the effect of a matrix function, such as $e^A v$, within
33 this low-dimensional subspace, where $m \ll n$. The subspace is constructed using
34 the Arnoldi or Lanczos algorithms, which iteratively compute an orthonormal basis
35 $V_m = [v_1, v_2, \dots, v_m]$ for $K_m(A, v)$ and a Hessenberg matrix $H_m = V_m^T A V_m$ that
36 represents the restriction of A to the subspace [4]. These algorithms are capable of
37 exploiting the sparsity of A , requiring only matrix-vector products and orthogonaliza-
38 tion steps, making it computationally efficient for large, sparse matrices typical in
39 PDE discretizations [4].

40 In the context of matrix exponentiation, the Krylov subspace method approxi-
41 mates

$$e^A v \approx V_m e_m^H e_1$$

42 where e_1 is the first standard basis vector, and H_m is a small $m \times m$ matrix whose
43 exponential can be computed efficiently using direct methods, such as Padé approx-
44 imation [9, 1]. This approach is particularly advantageous for problems like the PE
45 method, where the matrix A arises from the discretization of a differential operator
46 and is sparse (e.g., tridiagonal) [5]. The convergence of the Krylov approximation de-

48 pends on the spectral properties of A , with the required subspace dimension m being
 49 dependent on the spectral radius of A [7]. By reducing the problem to a smaller sub-
 50 space, Krylov methods achieve significant computational savings while maintaining
 51 high accuracy, making them well-suited for iterative range-stepping in wave propaga-
 52 tion models [7].

53 This paper explores the implementation and performance of Krylov subspace
 54 methods for matrix exponentiation in the PE solver, focusing on their application to
 55 underwater acoustics. By comparing the Krylov approach with traditional methods,
 56 such as Padé approximants, we highlight its advantages in handling complex sound
 57 speed profiles and large computational grids. From a logistical standpoint, all code
 58 used for this project was written in Python, with the scripts used to generate fig-
 59 ures shown in this report presented in the Appendices. For convenience, the project
 60 directory has also been made available on my personal Github. For all cases where
 61 there are no analytical exact solutions (such as for many of the PE applications),
 62 the "exact" solution was taken to be the output from Python built-in functions from
 63 the Scipy package. Other libraries used include the standard Numpy and Matplotlib
 64 packages, which were used for matrix handling and plotting, respectively.

65 The paper is organized as follows. Validation of the Python code is in section 2.
 66 An overview of the parabolic equation method for ocean acoustics is presented in
 67 section 3. A comparison to existing parabolic equation models for an isovelocity envi-
 68 ronment is in section 4, while a more realistic Munk profile is examined in section 5.
 69 Some numerical discussions regarding the choice in subspace dimension and computa-
 70 tional cost are presented in section 6 and section 7, respectively. Finally, a short
 71 conclusion is presented.

72 **2. Validation Using a 1D Heat Equation.** My own implementation of the
 73 Arnoldi iteration and the Krylov exponentiation formula shown in section 1 can be
 74 seen in Appendix A. We desire to show that our numerical solution works for simple
 75 partial differential equations prior to implementing a full oceanographic parabolic
 76 equation solver. To accomplish this, we seek to solve the the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

78 subject to the Dirichlet boundary conditions $u(0, t) = u(1, t) = 0$ and initial conditions
 79 $u(x, 0) = \sin(\pi x) + \frac{1}{2} \sin(3\pi x) + \frac{1}{10} \sin(5\pi x)$. It is trivial to show that the exact solution
 80 to this PDE is given by

$$81 \quad u(x, t) = e^{-\pi^2 \alpha t} \sin(\pi x) + \frac{1}{2} e^{-(3\pi)^2 \alpha t} \sin(3\pi x) + \frac{1}{5} e^{-(5\pi)^2 \alpha t} \sin(5\pi x)$$

82 The Krylov subspace method using a Dirichlet enforcement to the matrix (by
 83 accounting only for interior points) was implemented and used to solve this PDE
 84 using a finite difference scheme to approximate

$$85 \quad \frac{\partial u}{\partial t} = \left(\alpha \frac{\partial^2}{\partial x^2} \right) u$$

86 which has the exact solution of

$$87 \quad u_{i+1} = e^{A\Delta t} u_i$$

88 where A is the finite difference discretization of the second derivative operator and is
 89 tridiagonal.

90 The results for $m \in [1, 5]$ at $t = 0.5$ seconds are shown in Figure 1, which clearly
 91 show that for $m \geq 3$, our code recovers the exact analytic solution. This shows that
 92 our implementation of the Arnoldi iteration and Krylov approximation of the matrix
 93 exponentiation are performing as intended, providing confidence in the numerical
 94 implementation as we proceed to an ocean environment.

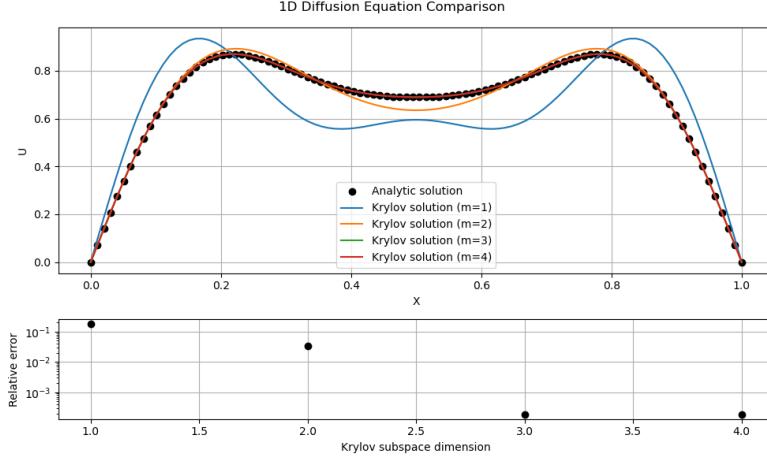


FIG. 1. Validation across different subspace dimensions for the 1D heat equation

95 **3. The Parabolic Equation.** The parabolic equation (PE) is a method of solving
 96 range-dependent acoustic boundary value problems in acoustics. To derive this,
 97 we first begin with the Helmholtz equation:

$$98 \quad \nabla^2 p + k^2 p = 0$$

99 where we must satisfy the Sommerfeld radiation condition ($p \rightarrow 0$ as $r \rightarrow \infty$), as well
 100 as boundary conditions at the surface and seabed interface. The typical boundary
 101 condition at the free surface is the pressure release boundary condition:

$$102 \quad p(z = 0) = 0$$

103 For the bottom, we typically enforce continuity of particle velocity. However, this
 104 implies that we are modeling sound propagation in the seabed. As the ocean-seafloor
 105 interface is not trivial to implement numerically, we will instead implement a rigid
 106 bottom boundary condition, which implies that:

$$107 \quad \left. \frac{\partial p}{\partial z} \right|_{z=D} = 0$$

108 We begin the derivation by separating the pressure field $p(x)$ into a slowly- and
 109 rapidly-varying function of range as:

$$110 \quad p(x) = \psi(x) H_0^{(1)}(k_0 r)$$

111 where the slowly varying contributions to the pressure field are covered by the spreading
 112 implied by the zeroth order Hankel function of the first kind and k_0 is a reference

113 wavenumber defined as ω/c_0 in terms of a reference sound speed (typically 1500 m/s)
 114 and the simulated angular frequency, ω . Note that the use of this Hankel function
 115 ensures that the final solution satisfies the Sommerfeld radiation condition. The new
 116 variable $\psi(x)$ is referred to as the envelope function and is what is being solved for in
 117 the PE method. The Helmholtz equation in terms of this envelope function can then
 118 be written as:

$$119 \quad \nabla^2\psi + i2k_0 \frac{\partial\psi}{\partial x} + k_0^2(n^2 - 1)\psi = 0$$

120 where n^2 is the reference index of refraction defined by $n^2 = (c_0/c)^2$. While the full
 121 derivation of the PE can be seen in textbooks (i.e. [5]), the simple explanation is that
 122 we factor the above equation, keeping only the forward propagating (i.e. outgoing)
 123 waves. This gives the exact parabolic equation as:

$$124 \quad \frac{\partial\psi}{\partial r} = ik_0 \left(-1 + \sqrt{n^2 + \frac{1}{k_0^2} \frac{\partial^2}{\partial z^2}} \right) \psi$$

125 It is common to denote the term involving the square root as Q in conjunction
 126 with a new operator $\mathcal{L} = ik_0(-1 + Q)$ such that the exact solution of the PE is given
 127 by:

$$128 \quad \psi(r + \Delta r) = e^{\Delta r \mathcal{L}} \psi(r)$$

129 It is precisely this equation that will be solved in this project using the Krylov sub-
 130 space matrix exponentiation technique validated in the previous section. We will
 131 also be computing this exponential directly using the `scipy.linalg.expm` function,
 132 which uses a Padé approximant of the exponentiation [1]. In the absence of analytic
 133 solutions, we will take the Scipy result to be the ground truth. Note here that Q is
 134 a class of fractional differential operator and the square root is not trivial to eval-
 135 uate in an exact sense. As such, many rational function approximations have been
 136 implemented for this "square root operator", with varying degrees of accuracy. The
 137 simplest approximation is the approximation introduced by Tappert [8] where:

$$138 \quad \sqrt{1 + q} \approx 1 + \frac{1}{2}q$$

139 where $q = n^2 - 1 + k_0^{-2}\partial_z^2$. Note that this is just the first order Taylor expansion
 140 of the square root. Other approximations exist, with the most commonly used in
 141 practice being the fifth order Padé expansion. The implementation of this method was
 142 considered outside the scope of this project, and instead the Tappert approximation
 143 was used for simplicity.

144 We now consider different implementations of the PE using the Tappert approx-
 145 imation in order to provide a comparison to commonly used formulations of this
 146 problem within the field of ocean acoustics. First, we consider the backward Euler
 147 method [5]. The backward euler solution can be expressed in terms of the \mathcal{L} oeprator
 148 as:

$$149 \quad \psi(r + \Delta r) = [1 - \Delta r \mathcal{L}]^{-1} \psi(r)$$

150 Using the Tappert approximation, the PE solution can be rewritten as.

$$151 \quad \psi(r + \Delta r) = \left[1 - \frac{1}{2}ik_0\Delta r \left(n^2 - 1 + \frac{1}{k_0^2} \frac{\partial^2}{\partial z^2} \right) \right]^{-1} \psi(r)$$

152 where it is clear that this matrix-vector multiplication allows for stepping in range.

153 Another commonly used method is the Crank-Nicholson (CN) method, which
 154 numerically is identical to a [1,1]-Padé approximation to e^A [9]. The CN solution to
 155 the PE can be written as:

$$156 \quad \left[1 - \frac{1}{4}ik_0\Delta r \left(n^2 - 1 + \frac{1}{k_0^2} \frac{\partial^2}{\partial z^2}\right)\right] \psi(r+\Delta r) = \left[1 + \frac{1}{4}ik_0\Delta r \left(n^2 - 1 + \frac{1}{k_0^2} \frac{\partial^2}{\partial z^2}\right)\right] \psi(r)$$

157 where left multiplying by the inverse of the bracketed term on the LHS allows for
 158 the range stepping of the pressure solution [5]. Having established the four different
 159 methods (i.e. `expm`, Krylov, CN, and Backward Euler) that will be used for evaluating
 160 PE performance, we can now analyze two simple PE environments. For all cases, we
 161 will utilize a range-independent sound speed profile (i.e. $c(r + \Delta r, z) = c(r, z)$). We
 162 will also implement only a 2D parabolic equation (which was what derived above).
 163 Finally, all simulations will be performed using a Gaussian starter for the initial
 164 condition of the envelope function defined as:

$$165 \quad \psi(0, z) = \sqrt{k_0} e^{-\frac{k_0^2}{2}(z-z_s)^2}$$

166 where z_s is the user-specified source depth [5]. Finally, the complex pressure field can
 167 be found by applying a simplified range decay (to approximate the Hankel function)
 168 as follows:

$$169 \quad p(r, z) = \frac{\psi(r, z)}{\sqrt{r}}$$

170 It is common then to display the transmission loss (TL) of the signal, which deter-
 171 mined how many decibels would be lost due to propagation and spreading loss. For
 172 our purposes, the transmission loss is calculated as:

$$173 \quad TL(r, z) = 20 \log_{10}(|p(r, z)|)$$

174 Note that typically this would be done to some reference pressure (typically 1 μPa
 175 for an ocean environment). Here we instead normalize by 1 Pa, which is the pressure
 176 magnitude 1 meter from the source location for an omnidirectional source (i.e. $p(1$
 177 meter)).

178 **4. The Isovelocity Case.** We first consider the case of a modified Pekeris wave-
 179 guide which does not account for the sound speed in the bottom and utilizing a pres-
 180 sure release boundary condition at the free surface and a rigid boundary condition at
 181 the seabed. Unfortunately, analytic solutions do not exist for this case¹ and as such,
 182 we will take the `expm` output (using a much reduced step size) as ground truth for
 183 model comparisons. Note that this is not perfect, as different step sizes may capture
 184 different interference effects, but it should still be elucidating as to the performance
 185 of the Krylov subspace methodology.

186 The sound speed profile was taken as a constant $c = c_0 = 1500$ m/s from the
 187 surface to a maximum depth of $D = 100$ meters, which is a simplified sound speed
 188 channel from what would be encountered in a continental shelf environment. The step
 189 size in depth, Δz , was taken to be 1 m. The simulation was performed for a $f = 20$
 190 Hz sound source located at $z_s = 36$ meters. Modeled TL for all four methodologies
 191 are shown in Figure 2, where the step size for the exact methodology was taken as
 192 $\Delta r/\lambda_0 = 1/3$ (i.e. $\Delta r = 5$ m), which should be sufficient to capture any significant

¹Exact modal solutions do exist for this environment; however, to limit the scope of this project, modal solutions were not numerically implemented.

interference patterns². It is clear that the backward Euler method performs the worst, failing to recreate any of the complex interference patterns captured by the other methods. We do note that the Krylov subspace methodology and the exact method match very closely. This becomes more clear when we look at a line plot of TL at the source depth between the four methods shown in Figure 3. Here we see that the Krylov subspace method is capable of almost perfectly recreating the TL when $m = 70$.

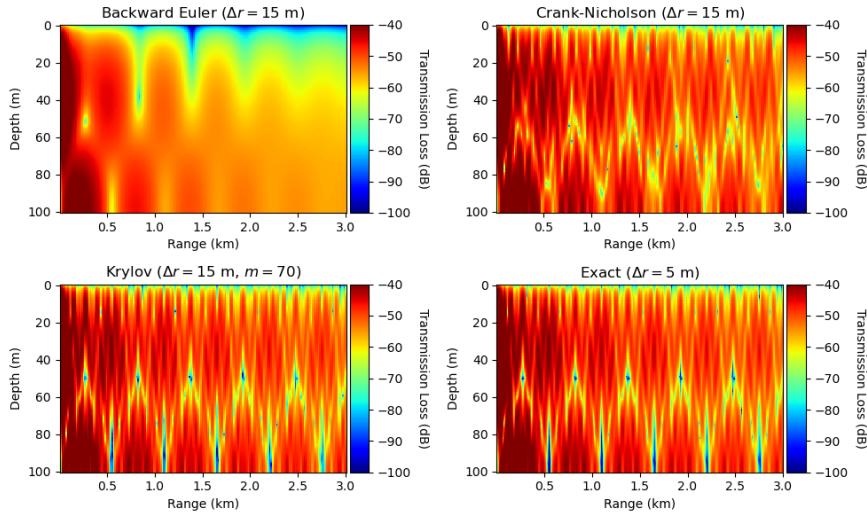


FIG. 2. Modelled transmission loss for the isovelocity case with $c = c_0 = 1500 \text{ m/s}$ and $D = 100 \text{ m}$

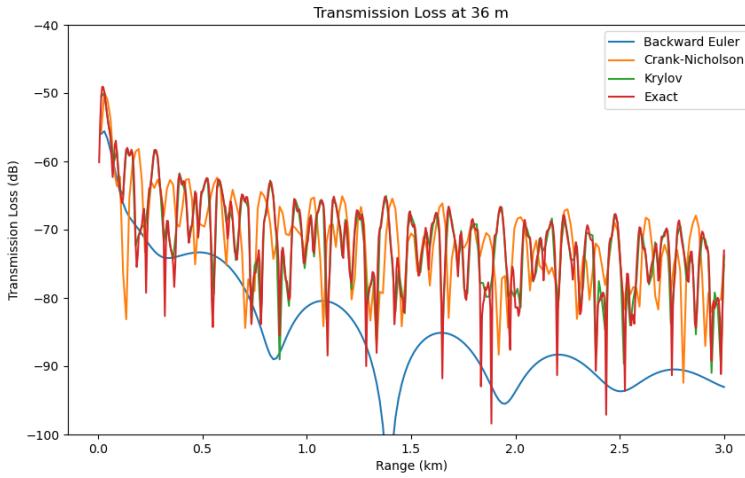


FIG. 3. Comparison of TL for the isovelocity case at the source depth of 36 m.

²For typical PE applications, the range marching step is taken to be $\sim 0.5\lambda_0 - 1\lambda_0$

200 **4.1. Sensitivity in Range Step and Subspace Dimension.** A sensitivity
 201 study was performed for the Krylov subspace method where the allowable range step,
 202 Δr , and Krylov subspace dimension, m , were varied and compared to the TL of
 203 the exact expm method. To provide a single value for comparison, the RMSE TL
 204 difference between the two TL curves was quantified by interpolating the denser,
 205 exact TL to the range steps of the Krylov subspace method. Note that one single
 206 value does not fully display the inaccuracies in each of the methods, as in reality we
 207 would need to compare the amplitude and phase of the complex pressure, as well as
 208 determine if certain methodologies are capable of simulating wider angle propagation
 209 (which would in turn lead to increased waveform interference). However, this full
 210 analysis was considered outside the scope of this project.

211 The results of this sensitivity analysis can be seen in Figure 4. This plot clearly
 212 shows that as the subspace dimension increases, larger range steps are possible for a
 213 given RMSE dB threshold. Note that the results shown in Figure 2 and Figure 3 were
 214 for $\Delta r/\lambda_0 = 1$, which suggests that we may have been able to use a smaller subspace
 215 dimension for similar accuracy in our initial simulation. This plot clearly shows the
 216 tradeoff in accuracy and computational efficiency (by reducing m and increasing Δr),
 217 which is relevant particularly for large domains.

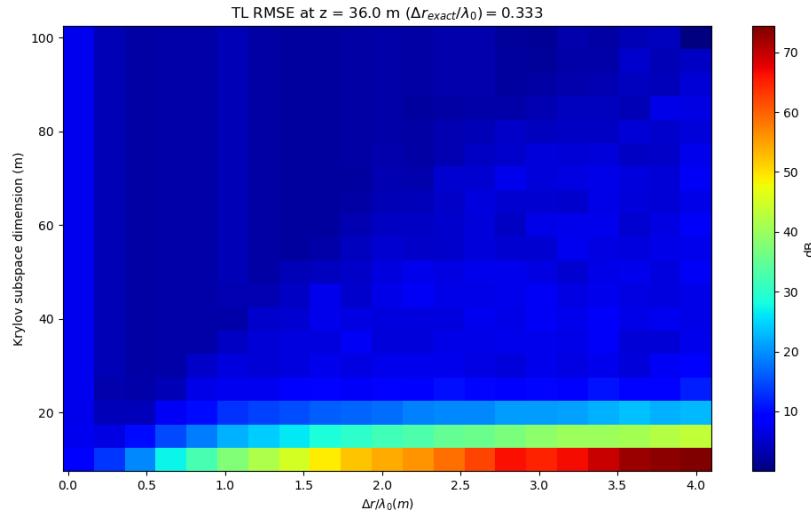


FIG. 4. *Sensitivity analysis of the Krylov subspace method in range step and subspace dimension for the isovelocity case.*

218 For comparison to the backward Euler and CN formulations, a sensitivity to Δr
 219 was performed. The results of this study can be seen in Figure 5. As before the step
 220 size for the exact methodology was taken as 5 meters. Here we see that the Krylov
 221 subspace method outperforms both of the other methods and is capable of achieving
 222 1 dB RMSE of the TL for $\Delta r_{Krylov} \approx 4\Delta r_{exact} = 1.3\lambda_0$. This means that for a given
 223 environment, we might be able to increase our range step by a factor of four while
 224 achieving similar accuracy for the prediction.

225 **5. The Munk Profile Case.** To simulate a more realistic, deep water ocean
 226 environment, another set of PE simulations were performed using the Munk profile.
 227 The Munk profile [6] is representative of a typical deep water sound speed profile and

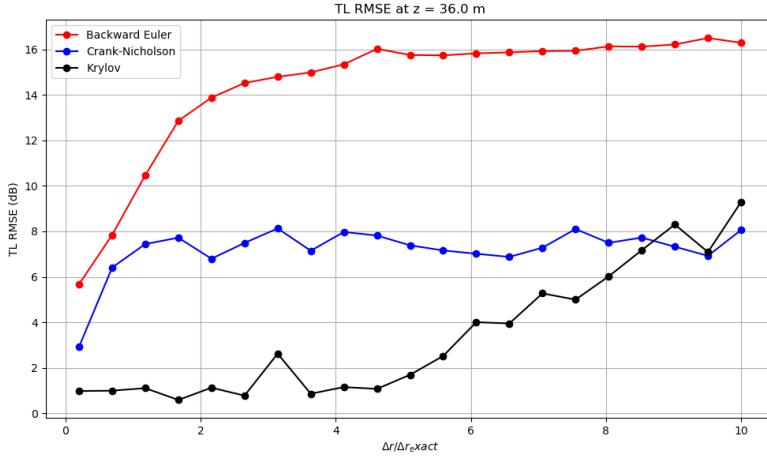


FIG. 5. Range step sensitivity for the isovelocity case with a Krylov subspace of dimension $m = 70$.

228 features a deep sound channel (i.e. SOFAR layer), which tends to trap energy inside
229 forming an acoustic duct. The Munk profile is defined analytically as:

$$230 \quad c(z) = 1500(1 + \epsilon(\bar{z} - 1 + e^{-\bar{z}}))$$

231 where $\bar{z} = 2(z - 1300)/1000$ and $\epsilon = 0.00737$. A visual representation of this sound
232 speed profile can be seen in Figure 6, where there is a clear axis located at $z = 1300$
233 m.

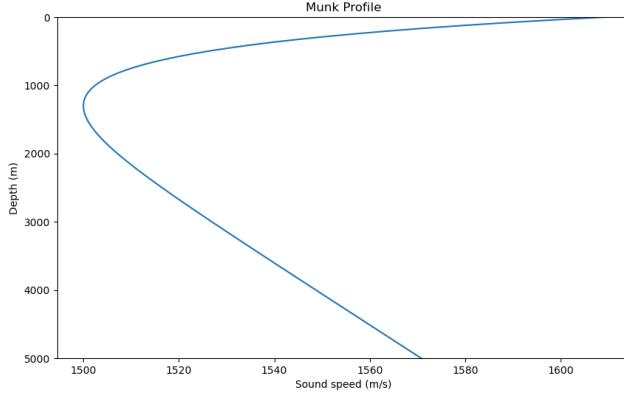


FIG. 6. Munk profile used in this project

234 PE models were performed for this profile with a 20 Hz source located at a source
235 depth of 1000 m. The maximum water depth was taken to be 5000 m with depth
236 discretized at 10 m intervals. Models were calculated using the backward Euler,
237 Crank-Nicholson, Krylov subspace, and `expm` method. The dimension of the Krylov
238 subspace was taken to be $m = 30$. All methods were simulated at a range step of 10
239 meters, which is less than the characteristic wavelength of 15 meters, to a maximum

range of 100 km in order to capture two cycles of the purely waterborne refraction in the SOFAR duct. The results of this simulation can be seen in Figure 7.

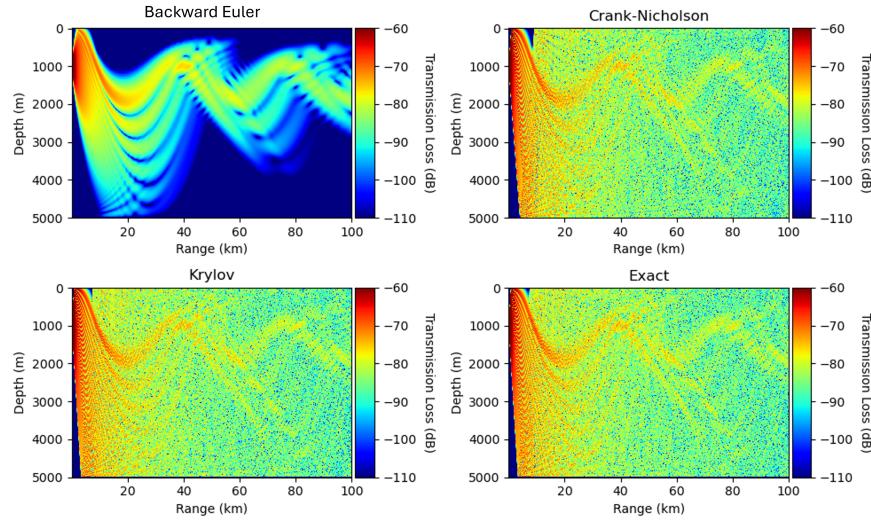


FIG. 7. Simulated transmission loss for a Munk profile in 5000 m water depth for a 20 Hz source located at 1000 m source depth.

Looking at the results, there are some interesting comparisons to be made between the four methods. First, there is no qualitative³ difference between the Krylov subspace method and the exact solution. This is interesting in that the Krylov subspace method provides significant cost savings over the exact method (see discussion in section 7 for a more detailed discussion). Second, from comparison between the CN and exact solutions, we see that the exact solution simulated a wider propagation angle than the CN method. This is evident from the size of the shadow zone (located at around 10 km in the near surface region), which is only insonified by reflected energy from the seabed. As this region is larger in the CN model, we can conclude that the exact (and therefore the Krylov method) must be able to model rays launched from the source at higher initial angles. In the extreme case, we see that the backward Euler method has much less noise, which is due to the inability of this method to model rays with even moderate launching angles. This leads to large shadow zones and significantly reduced interference patterns, as evident from the "less noisy" TL plot. All methods, however, are capable of capturing the primary propagation path within the sound duct, which is shown here by the sinusoidal patterns of lower transmission loss magnitude centered on the duct axis of 1300 m.

Unfortunately, due to the size of this domain, a full sensitivity analysis in Krylov subspace dimension and range step was not capable of being run. It is interesting that the Krylov subspace dimension of $m = 30$ is capable of still accurately recreating the TL field, especially when comparing to the isovelocity case, where $m = 70$ was necessary to fully capture the TL field with 1 dB RMSE. A more detailed discussion on why the dimension varies between the two cases is presented in section 6.

³There was also only minor quantitative differences; however, due to the scale of this domain I opted not to display line plots of the TL as it is very noisy and discrepancies were only evident when zooming to length scales on the order of 5 kilometers.

265 **6. Discussion on the Discrepancy in Subspace Dimension.** During our
 266 modeling efforts, it was seen that for the isovelocity case, the required subspace di-
 267 mension was $m = 70$ ($m_{max} = 100$) while for the Munk case the dimension was
 268 $m = 30$ ($m_{max} = 501$). This discrepancy primarily arises from our choice in the
 269 discretization of the sound speed profile. Recall that the matrix A being exponenti-
 270 ated is effectively the finite difference matrix for the second derivative in z with an
 271 additional diagonal term accounting for the squared index of refraction.

$$272 \quad A = ik_0\Delta r(D + n^2 - 1)$$

273 where D represents the discretized second derivative.

274 For the isovelocity case, $n^2 = 1$ which causes the matrix to be purely the finite
 275 difference discretization, which scales with Δz^{-2} . If we consider the spectral radius
 276 of A (the largest eigenvalue of A), we see that it too scales with Δz^{-2} . The same
 277 holds true for the Munk profile case, as the leading term in the eigenvalues of A
 278 is still the finite difference matrix. In effect, this means that the spectral radius is
 279 determined primarily by choice of Δz . Computing the largest eigenvalues directly
 280 for our exponents shows that the maximum eigenvalue was $-2.42i$ and $-358i$ for the
 281 Munk and isovelocity cases, respectively.

282 Based on discussion from Refs. [7] and [4], we see that the "optimal" value
 283 of m is dependent on convergence of the polynomial representation of A and by
 284 extension dependent on the magnitude of the spectral radius. As such, it is clear
 285 that our selection of a small Δz for the isovelocity case amplifies the spectrum of
 286 D . Increasing the spectral radius increases the degree of the polynomial needed to
 287 accurately approximate the matrix A , which implies that a larger Krylov subspace
 288 dimension is necessary in order to capture the effect of e^A on the envelope function
 289 ψ . This was an oversight when I was running the simulations; however, due to time
 290 constraints the analysis was not able to be rerun with a larger depth grid size.

291 **7. Discussion on the Computational Cost.** In this section, we will discuss
 292 the computational cost associated with the methods discussed above. As the Crank-
 293 Nicholson and forward Euler methods exhibited large errors, we will not consider
 294 them for this comparison, as the best performing model was the Krylov subspace
 295 methodology.

296 We first consider the "exact" solution provided by directly computing $e^A\psi_i$ using
 297 `scipy.linalg.expm`. This function uses an algorithm for the matrix exponentiation
 298 using a Padé approximant to approximate the direct solution [1]. First, A^k is com-
 299 puted, which for $p = q = 6$ results in $12n_z^3$ FLOPs. The determination of $N_{p,q}(A)$ and
 300 $D_{p,q}(A)$ costs an additional $12n_z^3$ FLOPs. Solving the system $D_{p,q}(A)x = N_{p,q}(A)$
 301 results in $\frac{2}{3}n_z^3$ FLOPs for the LU decomposition and $2n_z^3$ FLOPs for the back substi-
 302 tution step. In total, the computation of the Padé approximant results in $\frac{44}{3}n_z^3 + 12n_z^2$
 303 FLOPs. The scaling and squaring step proposed by Higham results in an additional
 304 $8n_z^3$ FLOPs assuming that the result is squared four times during the scaling and
 305 squaring step. Finally, we multiply this approximant of e^A by the initial vector ψ_i .
 306 This results in an additional $2n_z^2$ FLOPs. As such, the total computational cost of
 307 the "exact" method is $\frac{68}{3}n_z^3 + 14n_z^2$, which to leading order is $\sim 22n_z^3$ FLOPs [7].

308 In contrast, consider the Krylov methodology. The Arnoldi iteration is performed
 309 m times, for a total cost of $2mn_z(m + 5) + 20m$ for a tridiagonal matrix. The matrix
 310 exponentiation $e^H e_1$ is computed using a Padé approximant for a total cost of $\frac{2}{3}m^3 +$
 311 $14m^2$. Finally, the result is multiplied by the initial vector for an additional cost of
 312 $(2n_z + 1)m + 2n_z$ operations. In summary, we see that to leading order the Krylov

313 exponentiation for a tridiagonal A , as in our case, costs $\sim 2n_z m^2$, whereas the direct
 314 method costs $\sim 23n_z^3$. This analysis matches closely with the analysis of [7].

315 As such, it is clear that the Krylov exponentiation method is much more computationally
 316 efficient than the Padé approximant since $m \ll n_z$ as shown by the results
 317 of our m sweep analysis. As illustration, for the Munk profile case with $n_z = 501$ and
 318 $m = 30$, we see that the exact method costs $\mathcal{O}(2.8 \times 10^9)$ FLOPs whereas the Krylov
 319 method costs $\mathcal{O}(9 \times 10^5)$ FLOPs. Since this cost would be done at each range step, it
 320 is clear that using the Krylov exponentiation method is more computationally efficient
 321 particularly in range-dependent cases where e^A would need to be recomputed at each
 322 step. Note that for range-independent problems, this advantage is not as pronounced,
 323 since the Arnoldi iteration is reliant on the initial vector, ψ_i , and would need to be to
 324 be computed at each range step, whereas a range-independent A would not need to
 325 be recomputed for the exact methodology.

326 It is also worth comparison to the "epade" approximation utilized in the Range-
 327 Dependent Acoustic Model (RAM) PE developed by Michael Porter, which is the
 328 most commonly used variant of the PE for ocean acoustic problems [3]. This ap-
 329 proximation utilizes tandem approximations for both the square root approximation
 330 (i.e. the Tappert approximation used in this project) and for the matrix exponentia-
 331 tion in what is referred to as the split-step Padé methodology [2]. This methodology
 332 is typically used with a low number of Padé coefficients needing to be solved to meet
 333 operational accuracy constraints (i.e. $n_p \approx 5$). In total, for a single range-step, this
 334 "epade" method costs $\mathcal{O}(n_z n_p)$. This provides significant cost savings due to the tai-
 335 loring of this methodology to the solution of PE methods and explains why it is the
 336 most commonly used solver for the PE. A summary of the costs for each of the three
 337 methods investigated here is shown in Table 1. It is clear that while both the "epade"
 338 and Krylov methods scale as $\mathcal{O}(n_z)$, the coefficient associated with this computational
 339 cost is many orders of magnitude larger for the Krylov method when compared to
 340 that established by Collins.

Method	Cost (Approx.)	Munk Cost
<code>expm</code> [1]	$\mathcal{O}(22n_z^3)$	$\sim 2.8 \times 10^9$
Krylov Subspace	$\mathcal{O}(2n_z m^2) \approx \mathcal{O}(1800n_z)$	$\sim 9 \times 10^5$
<code>epade</code> (Collins)	$\mathcal{O}(n_z n_p) \approx \mathcal{O}(5n_z)$	$\sim 2.5 \times 10^3$

TABLE 1
Cost summary (in FLOPs) of the three methods discussed.

341 For the case of the simplified Munk profile using our empirically determined values
 342 of $m = 30$ and $n_z = 501$, it is clear that while the Krylov subspace method investigated
 343 in this project provides significant cost benefits over the direct exponentiation built-in
 344 using the `expm` function, it does not provide cost savings over the traditional method
 345 developed by Collins [2]. In particular, the most costly portion of the Krylov subspace
 346 method is the Arnoldi iteration, where the orthogonalization is the leading term in
 347 the overall cost with $\mathcal{O}(n_z m^2)$ operations needed to orthogonalize the bases against
 348 those previously determined. Note that while these costs would be incurred at each
 349 range step (adding another factor of n_r into the total cost), it was found for the
 350 isovelocity case that the maximum $\Delta r_{Krylov}/\Delta r_{expm}$ was around $\mathcal{O}(2 - 3)$. If we
 351 assume that similar range step increases can be implemented when comparing "epade"
 352 to the Krylov subspace (since both "epade" and Higham's method are based on Padé
 353 approximants), this factor is still negligible compared to the increase in cost incurred

354 by the m^2 in the Krylov subspace leading order cost. While the "epade" methodology
 355 implemented by Collins is only valid for PE applications, the Krylov subspace and
 356 Higham expm are more general, meaning that in the absence of tailor built algorithms
 357 to solve this Matrix exponentiation, the Krylov subspace is the superior method for
 358 general applications.

359 **8. Conclusions.** In this report, we discussed the Krylov subspace method for
 360 approximating the matrix exponential. Code was developed in Python implementing
 361 the Krylov approximation and the Arnoldi iteration. The code was validated against
 362 a one-dimensional heat equation solution with non-trivial initial conditions before being
 363 used to simulate realistic ocean environments. Ocean environments were modeled
 364 using the parabolic equation method with the backwards Euler, Crank-Nicholson,
 365 Krylov subspace, and direct matrix exponentiation being implemented. The first case
 366 was a shallow water, isovelocity environment, where it was demonstrated that the
 367 Krylov subspace method is capable of accurately recreating the exact solution. A
 368 sensitivity analysis on the range step and the subspace dimension was performed that
 369 showed the Krylov subspace is capable of both reducing the dimensionality of the
 370 exponent term (e.g. from $m = 100$ to $m = 70$), as well as increasing the range step
 371 by approximately a factor of three. The second environment was a representative
 372 deep water environment, characterized by the Munk profile. In this case, the Krylov
 373 subspace method was capable of recreating the exact solution while significantly re-
 374 ducing the subspace dimension from $m = 501$ to $m = 30$. A discussion on why these
 375 two cases had different subspace dimensions was presented, where it was concluded
 376 that the primary difference between them was the depth grid discretization scheme.
 377 A computation cost analysis was performed for each of the four methods, as well as
 378 a comparison to the exponential Padé approximation commonly used in operational
 379 acoustic codes. It was found that the Krylov subspace method provides both sig-
 380 nificant cost savings over the direct exponentiation and increased accuracy over the
 381 backwards Euler and Crank-Nicholson solution. Comparison to Collins' [3] expon-
 382 ential Padé implementation, showed that both it and the Krylov subspace method scale
 383 as $\mathcal{O}(n_z)$. However, the coefficient for the cost in the Krylov subspace was orders of
 384 magnitude larger than that of Collins' implementation. As such, it was concluded
 385 that while the Krylov subspace method provides cost savings over direct methods, it
 386 does not favorably compare to the established state-of-the-art algorithms currently
 387 implemented in standard ocean acoustics parabolic equation solvers.

- 389 [1] A. H. AL-MOHY AND N. J. HIGHAM, *A New Scaling and Squaring Algorithm for the Matrix*
 390 *Exponential*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 970–989,
 391 <https://doi.org/10.1137/09074721X>.
- 392 [2] M. D. COLLINS, *A split-step Padé solution for the parabolic equation method*, The Journal of the
- 393 Acoustical Society of America, 93 (1993), pp. 1736–1742, <https://doi.org/10.1121/1.406739>.
- 394 [3] M. D. COLLINS, *User's Guide for RAM Versions 1.0 and 1.0 p*, Naval Research Lab, Washington,
- 395 DC, 20375 (1995), p. 14.
- 396 [4] N. J. HIGHAM, *Functions of Matrices: Theory and Computation*, Society for industrial and
- 397 applied mathematics, Philadelphia, 2008.
- 398 [5] F. B. JENSEN, W. A. KUPERMAN, M. B. PORTER, AND H. SCHMIDT, *Computational Ocean Acous-*
- 399 *tics*, Springer New York, New York, NY, 2011, <https://doi.org/10.1007/978-1-4419-8678-8>.
- 400 [6] W. H. MUNK, *Sound channel in an exponentially stratified ocean, with application to SOFAR*,
- 401 The Journal of the Acoustical Society of America, 55 (1974), pp. 220–226, <https://doi.org/10.1121/1.1914492>.
- 402 [7] Y. SAAD, *Analysis of Some Krylov Subspace Approximations to the Matrix Exponential Opera-*

- 404 tor, SIAM Journal on Numerical Analysis, 29 (1992), pp. 209–228, <https://doi.org/10.1137/0729014>.
- 405 [8] F. D. TAPPERT, *The parabolic approximation method*, in Wave Propagation and Underwater
406 Acoustics, J. B. Keller and J. S. Papadakis, eds., vol. 70, Springer Berlin Heidelberg, Berlin,
407 Heidelberg, 1977, pp. 224–287, https://doi.org/10.1007/3-540-08527-0_5.
- 408 [9] H. WANG, *The Krylov Subspace Methods for the Computation of Matrix Exponentials*, Dec.
409 2015.
- 410

411 **Appendix A. Code for Krylov Exponentiation.** The following code was
412 used for the Arnoldi iteration and Krylov subspace exponentiation method. A full
413 project directory can be found at https://github.com/harriswr/krylov_pe.

```
414 1 """Module to house matrix exponential and Krylov subspace methods."""
415 2
416 3 from scipy.linalg import expm, norm
417 4 import numpy as np
418 5 import numpy.typing as npt
419 6
420 7
421 8 def arnoldi_iteration(
422 9     input_mat: npt.NDArray,
423 10    init_vector: npt.NDArray,
424 11    num_iterations: int,
425 12 ) -> tuple[npt.NDArray, npt.NDArray]:
426 13     """Perform m steps of Arnoldi iteration to construct the subspace.
427 14
428 15     Parameters
429 16     -----
430 17     input_mat : npt.ndarray
431 18         The matrix for which we want to compute the Krylov subspace.
432 19     init_vector : npt.ndarray
433 20         The initial vector to start the Arnoldi iteration.
434 21     num_iterations : int
435 22         The number of steps of the Arnoldi iteration.
436 23
437 24     Returns
438 25     -----
439 26     basis_mat : npt.ndarray
440 27         The orthonormal basis of the Krylov subspace.
441 28     hessen_mat : npt.ndarray
442 29         The upper Hessenberg matrix that represents the action of A on
443 30         the Krylov subspace.
444 31
445 32 """
446 33     if isinstance(input_mat, np.matrix):
447 34         input_mat = np.asarray(input_mat)
448 35
449 36     n = input_mat.shape[0]
450 37     m = num_iterations
451 38     basis_mat = np.zeros(
452 39         (n, num_iterations), dtype=input_mat.dtype
453 40     ) # Orthonormal basis of the Krylov subspace
454 41     hessen_mat = np.zeros((m, m), dtype=input_mat.dtype) # Upper
455 42     Hessenberg matrix
456 43
457 44     # normalize the initial vector
458 45     v_norm = norm(init_vector)
459 46     if v_norm < 1e-10:
460 47         print("Warning: Initial vector norm is near zero in Arnoldi
461 48         iteration")
462 49     init_vector = init_vector / v_norm
463 50     basis_mat[:, 0] = init_vector
464 51
```

```

46549 for j in range(m - 1):
46650     w = input_mat @ basis_mat[:, j] # w = A * v_j
46751
46852     # orthogonalize against the previous bases
46953     for i in range(j + 1):
47054         hessen_mat[i, j] = np.vdot(basis_mat[:, i], w)
47155         w -= hessen_mat[i, j] * basis_mat[:, i]
47256
47357     # normalize and store
47458     hessen_mat[j + 1, j] = norm(w)
47559     if hessen_mat[j + 1, j] > 1e-10:
47660         basis_mat[:, j + 1] = w / hessen_mat[j + 1, j]
47761     else:
47862         return basis_mat, hessen_mat
47963
48064 w = input_mat @ basis_mat[:, m - 1] # w = A * v_m-1
48165 for i in range(m):
48266     hessen_mat[i, m - 1] = np.vdot(basis_mat[:, i], w)
48367     w -= hessen_mat[i, m - 1] * basis_mat[:, i]
48468
48569 return basis_mat, hessen_mat
48670
48771
48872 def krylov_expm(
48973     input_mat: npt.NDArray,
49074     init_vector: npt.NDArray,
49175     num_iterations: int,
49276 ) -> npt.NDArray:
49377     """Approximate the matrix exponential using the Krylov subspace.
49478
49579     Parameters
49680     -----
49781     input_mat : npt.ndarray
49882         The matrix for which we want to compute the Krylov subspace.
49983     init_vector : npt.ndarray
50084         The initial vector to start the Arnoldi iteration.
50185     num_iterations : int
50286         The number of steps of the Arnoldi iteration and also the
50387         dimension of the Krylov subspace.
50488
50589     Returns
50690     -----
50791     new_vector : npt.ndarray
50892         The result of the matrix exponential applied to the initial
50993         vector.
51094         This is the approximation of expm(input_mat) @ init_vector.
51195     """
51296
51397     # Perform Arnoldi iteration
51498     basis_mat, hessen_mat = arnoldi_iteration(input_mat, init_vector,
51599     num_iterations)
516100
517101     # Compute the matrix exponential of the upper Hessenberg matrix
518102     beta = norm(init_vector)
519103     e1 = np.zeros(hessen_mat.shape[0])
520104     e1[0] = 1.0 # First basis vector
521105     expm_hessen = expm(hessen_mat)
522106
523107     return beta * basis_mat @ (expm_hessen @ e1)
524108
525109
526110
527111 def krylov_expm_with_dirichlet(

```

```

527 08     input_mat: npt.NDArray,
528 09     idx: npt.NDArray,
529 10     init_vector: npt.NDArray,
530 11     num_iterations: int,
531 12     total_number_of_points: int,
532 13 ) -> npt.NDArray:
533 14     # pull only the interior matrices
534 15     next_mat = np.zeros(total_number_of_points)
535 16     next_interior = krylov_expm(input_mat, init_vector[idx],
536 17     num_iterations)
537 18     next_mat[idx] = next_interior
538 19     return next_mat

```

539 **Appendix B. Code for Diffusion Validation.** The following code was a
 540 module used in the diffusion validation case.

```

541 1 """Module to use for 1D heat equation validation cases."""
542 2
543 3 import numpy as np
544 4 import numpy.typing as npt
545 5 from scipy.sparse import diags
546 6
547 7
548 8 def create_1d_diffusion_matrix(
549 9     n: int, dx: float, alpha: float
550 0 ) -> tuple[npt.NDArray, npt.NDArray]:
551 1     """Create a 1D diffusion matrix for the heat equation.
552 2
553 3     Assumes that x is discretized on a uniform grid with spacing dx =
554 4     1/(n-1)
555 5     with Dirichlet boundary conditions at x=0 and x=1.
556 6
557 7     Parameters
558 8     -----
559 9     n : int
560 0     Number of spatial points.
561 0     dx : float
562 1     Spatial discretization step size.
563 2     alpha : float
564 3     Diffusion coefficient.
565 4
566 5     Returns
567 6     -----
568 7     diff_mat: npt.NDArray
569 8         The diffusion matrix.
570 9     idx: npt.NDArray
571 0         The indices of the interior points.
572 1
573 2     n_interior = n - 2 # Number of interior points
574 3     diag = -2 * alpha / (dx**2) * np.ones(n_interior)
575 4     off_diag = alpha / (dx**2) * np.ones(n_interior - 1)
576 5     diff_mat = diags([off_diag, diag, off_diag], [-1, 0, 1]).toarray()
577 6     # type: ignore[no-untyped-call]
578 6
579 7     # indices for the interior points
580 8     idx = np.arange(1, n - 1)
581 9
582 0     return diff_mat, idx
583 1
584 2
585 3 def analytic_1d_diffusion_solution(
586 4     x: npt.NDArray,

```

```

58745     t: float,
58846     alpha: float,
58947 ) -> npt.NDArray:
59048     """Compute the analytic solution of the 1D diffusion equation.
59149
59250     Analytic solution for 1D diffusion equation with initial condition
59351     u(x,0) = sin(pi*x) + 0.5*sin(3*pi*x) + 0.2*sin(5*pi*x).
59452
59553     Assumes dirchilet boundary conditions at x=0 and x=1.
59654     Parameters
59755     -----
59856     x : npt.NDArray
59957         Spatial points.
60058     t : float
60159         Time point.
60259     alpha : float
60359         Diffusion coefficient.
60459     initial_condition : npt.NDArray | None, optional
60559         Initial condition. Default is None, in which case a default
60659         intial condition of u(x,0) = sin(pi*x) is used.
60759
60859     Returns
60959     -----
61059     npt.NDArray
61159         The analytic solution at time t.
61259
61359     return (
61459         np.exp(-(np.pi**2) * alpha * t) * np.sin(np.pi * x)
61559         + 0.5 * np.exp(-((3 * np.pi) ** 2) * alpha * t) * np.sin(3 * np.
61659         pi * x)
61759         + 0.2 * np.exp(-((5 * np.pi) ** 2) * alpha * t) * np.sin(5 * np.
61859         pi * x)
61959     )

```

620 The following code was the script used to generate the 1D heat validation plots.

```

6211 """Script to create validation against analytic solution for 1D
6221     diffusion equation."""
6231
6243 import numpy as np
6254 import matplotlib.pyplot as plt
6265 from scipy.linalg import norm
6276
6287 from krylov_pe.krylov import krylov_expm_with_dirichlet
6298 from krylov_pe.diffusion import (
6309     create_1d_diffusion_matrix,
6310     analytic_1d_diffusion_solution,
6321 )
6332
6343
6354 def main():
6365     """Validate the Krylov subspace method for 1D diffusion equation."""
6376
6387     # Parameters
6398     n = 100 # Number of spatial points
6409     alpha = 0.01 # Diffusion coefficient
6410     t = 0.5 # Time step
6421
6432     # create the grid and intitial conditions
6443     x = np.linspace(0, 1, n, dtype=np.float64) # Spatial grid
6454     dx = x[1] - x[0]
6465     u0 = (
6476         np.sin(np.pi * x) + 0.5 * np.sin(3 * np.pi * x) + 0.2 * np.sin(5

```

```

648     * np.pi * x)
649 ) # Initial condition
650
651 # analytic solution
652 u_analytic = analytic_1d_diffusion_solution(x, t, alpha)
653
654 # Plot
655 fig = plt.figure(figsize=(10, 6))
656 ax = fig.add_subplot(3, 1, (1, 2))
657 ax2 = fig.add_subplot(3, 1, 3)
658 ax.plot(x, u_analytic, "ko", label="Analytic solution")
659
660 for m in np.arange(1, 5, 1, dtype=int): # Dimension of Krylov
661     subspace
662         # Create the diffusion matrix
663 diff_mat, idx = create_1d_diffusion_matrix(n, dx, alpha)
664 u_krylov = krylov_expm_with_dirichlet(diff_mat * t, idx, u0, m,
665 n)
666
667 ax.plot(x, u_krylov, label=f"Krylov solution (m={int(m)})")
668
669 # error
670 error = norm(u_krylov - u_analytic) / norm(u_analytic)
671 ax2.plot(m, error, "ko")
672
673 ax.set_xlabel("X")
674 ax.set_ylabel("U")
675 ax.legend()
676 ax.grid()
677
678 ax2.set_xlabel("Krylov subspace dimension")
679 ax2.set_ylabel("Relative error")
680 ax2.set_yscale("log")
681 ax2.grid()
682 fig.suptitle("1D Diffusion Equation Comparison")
683 plt.tight_layout()
684 plt.show(block=True)
685
686
687 if __name__ == "__main__":
688     main()

```

689 **Appendix C. Functions for Parabolic Equation.** The following code was
690 the base module implementing the common parabolic equation solvers.

```

691 """Module for PE comparison."""
692
693 from matplotlib.axes import Axes
694 import numpy as np
695 import numpy.typing as npt
696 import scipy.linalg as linalg
697 from scipy.sparse import diags
698 from scipy.linalg import expm
699
700 from krylov_pe.krylov import krylov_expm
701
702
703 def munk_profile(z: np.ndarray) -> np.ndarray:
704     """Munk profile for oceanographic applications.
705
706     Parameters
707     -----

```

```

7088     z : np.ndarray
7099         Depth array.
7100
7111     Returns
7122     -----
7133     np.ndarray
7144         Munk profile.
7155
7166     Reference:
7177     W. H. Munk, "Sound channel in an exponentially stratified ocean with
718     applications to SOFAR,"
7198     J. Acoust. Soc. Am. 55, 220--226 (1974).
7209
7210     eps = 0.00737
7221     z_bar = 2 * (z - 1300) / 1000
7232     return 1500 * (1 + eps * (z_bar - 1 + np.exp(-z_bar)))
7243     # eta = 2 * (z - 1000) / 1000 # Normalized depth
7254     # return 1500 * (1 + 0.0057 * (-1 + eta + np.exp(-eta))) # Munk
726
7275
7286
7297 def build_depth_grid(zmax: int = 5000, dz: int = 10) -> np.ndarray:
7308     """Build a depth grid for oceanographic applications.
7319
7320     Returns
7331     -----
7342     np.ndarray
7353         Depth grid.
7364
7375     h = zmax
7386     return np.arange(0, h + dz, dz) # Depth grid
7397
7408
7419 def gaussian_starter(k0: float, z_src: float, z: npt.NDArray[np.float64]
7420 ) -> np.ndarray:
7431     """Gaussian starter for oceanographic applications.
7442
7453     Parameters
7464     -----
7475     k0 : float
7485         Reference wavenumber (1/m).
7496     z_src : float
7507         Source depth (m).
7518     z : np.ndarray
7529         Depth array.
7530
7541     Returns
7552     -----
7563     np.ndarray
7574         Gaussian starter.
7585
7596     return np.sqrt(k0) * np.exp(-(k0**2) / 2 * (z - z_src) ** 2) # #
760
7617
7628
7639 def make_finite_difference_matrix(
7640     z: npt.NDArray[np.float64],
7651 ) -> npt.NDArray[np.float64]:
7662     """Make finite difference matrix for 2nd derivative in z."""
7673     n = len(z)
7684     dz = z[1] - z[0] # Depth step (m)
7695

```

```

770r6 diag = -2 * np.ones(n) / (dz**2) # Diagonal
771r7 off_diag = np.ones(n - 1) / (dz**2) # Off-diagonal
772r8 diff_mat = diags([off_diag, diag, off_diag], [-1, 0, 1]).toarray()
773# type: ignore[no-untyped-call]
774r9 diff_mat[-1, -1] = -1 # enforces the rigid boundary condition at
775the bottom
776r0 return diff_mat # Finite difference matrix
777r1
778r2
779r3 def propagator_backward_euler(
780r4     reference_wavenumber: float,
781r5     z: npt.NDArray,
782r6     r: npt.NDArray,
783r7     n_squared: npt.NDArray,
784r8     diff_mat: npt.NDArray,
785r9 ) -> np.ndarray:
786r0     """Propagator for backward Euler method."""
787r1     nz = len(z)
788r2     dr = r[1] - r[0]
789r3
790r4     propagator_matrix = np.eye(nz) - 1j / 2 * reference_wavenumber * dr
791r5     * (
792r6         diags(n_squared - 1) + diff_mat / reference_wavenumber**2
793r7     )
794r8
795r9     return linalg.inv(propagator_matrix)
796r0
797r0
798r1 def propagator_crank_nicholson(
799r2     reference_wavenumber: float,
800r3     z: npt.NDArray,
801r4     r: npt.NDArray,
802r5     n_squared: npt.NDArray,
803r6     diff_mat: npt.NDArray,
804r7 ) -> np.ndarray:
805r8     """Propagator for Crank-Nicholson method."""
806r9     nz = len(z)
807r0     dr = r[1] - r[0]
808r1
809r2     propagator_matrix_1 = np.eye(nz) - 1j / 4 * reference_wavenumber *
810r3     dr * (
811r4         diags(n_squared - 1) + diff_mat / reference_wavenumber**2
812r5     )
813r6     propagator_matrix_2 = np.eye(nz) + 1j / 4 * reference_wavenumber *
814r7     dr * (
815r8         diags(n_squared - 1) + diff_mat / reference_wavenumber**2
816r9     )
817r0
818r1     return linalg.inv(propagator_matrix_1) @ propagator_matrix_2
819r0
820r1
821r2 def simple_parabolic_equation(
822r3     freq: float,
823r4     z: npt.NDArray[np.float64],
824r5     z_src: float,
825r6     r_max: float,
826r7     dr: float,
827r8     ssp: npt.NDArray[np.float64],
828r9     propagator: str,
829r0 ) -> tuple[np.ndarray, np.ndarray]:
830r1     """Simple parabolic equation for oceanographic applications.
831r2

```

```

8323     Returns
8324     -----
8325     np.ndarray
8326         Simple parabolic equation.
8327     """
8328     # depth grid
8329     wavenumber = 2 * np.pi * freq / ssp
8330
8431     # range grid
8432     r = np.arange(dr, r_max + dr, dr) # Range grid
8433
8434     # initialize the PE matrix
8435     psi = np.zeros((len(z), len(r)), dtype=np.complex128) # PE matrix
8436
8437     # reference values for PE step
8438     ref_ssp = 1500 # Reference sound speed (m/s)
8439     ref_wavenumber = 2 * np.pi * freq / ref_ssp # Reference wavenumber
8440     (1/m)
8541     n_squared = (wavenumber / ref_wavenumber) ** 2
8542
8543     # starter
8544     psi0 = gaussian_starter(ref_wavenumber, z_src, z) # Gaussian
8545     starter
8546
8547     # farfield bessel function
8548     d_sq = make_finite_difference_matrix(z) # Finite difference matrix
8549
8550     match propagator:
8651         case "backward_euler":
8652             pe_propagator = propagator_backward_euler(
8653                 ref_wavenumber, z, r, n_squared, d_sq
8654             )
8655         case "crank_nicholson":
8656             pe_propagator = propagator_crank_nicholson(
8657                 ref_wavenumber, z, r, n_squared, d_sq
8658             )
8659         case _:
8660             raise ValueError(
8661                 "Invalid propagator type. Use 'backward_euler' or "
8662                 "crank_nicholson'."
8663             )
8764
8765     psi[:, 0] = pe_propagator @ psi0
8766     for i in np.arange(0, len(r) - 1, dtype=int):
8767         psi[:, i + 1] = pe_propagator @ psi[:, i]
8768
8769     # apply bessel function for range decay
8770     hk = 1 / np.sqrt(r)
8771     press = psi * hk
8772
8773     return press, r
8874
8875 def press_to_db(press: npt.NDArray[np.complex128]) -> npt.NDArray[np.
8876 float64]:
8877     """Convert pressure to dB"""
8878     return 20 * np.log10(np.abs(press))
8879
8880
8981 def plot_pe_result(
8982     ax: Axes,
8983     r: npt.NDArray[np.float64],

```

```

8941     z: npt.NDArray[np.float64],
8952     press: npt.NDArray[np.complex128],
8963     title: str,
8974     clims: tuple[float, float] = (-110, -60),
8985 ):
8996     """Plot the transmission loss field"""
9007     r_km = r / 1e3 # Convert range to km
9018     p_db = press_to_db(press) # Convert pressure to dB
9029
9030     ax.set_title(title)
9041     ax.set_xlabel("Range (km)")
9052     ax.set_ylabel("Depth (m)")
9063     ax.set_aspect("auto")
9074     ax.invert_yaxis()
9085     mesh = ax.pcolormesh(r_km, z, p_db, shading="auto", cmap="jet")
9096     mesh.set_clim(clims[0], clims[1]) # Set color limits for dB scale
9107     cbar = ax.get_figure().colorbar(mesh, ax=ax, pad=0.01, aspect=10) #
9118         type: ignore
9128     cbar.set_label("Transmission Loss (dB)", rotation=270, labelpad=15)
9139
9140
9151 def plot_pe_tl_at_z(
9162     ax: Axes,
9173     r: npt.NDArray[np.float64],
9184     z: npt.NDArray[np.float64],
9195     press: npt.NDArray[np.complex128],
9206     z_plt: float,
9217     label: str,
9228     dr_plot: float = 100,
9239 ):
9240     """Plot line plot of transmission loss at a given depth."""
9251     dr = r[1] - r[0] # Range step (m)
9262     r_skip_ind = int(dr_plot / dr) # Skip every r_skip_ind points
9273     r_skip_ind = max(1, r_skip_ind) # Ensure at least one point is
9284     plotted
9295     r = r[::-r_skip_ind] # Skip points for plotting
9306
9317     r_km = r / 1e3 # Convert range to km
9328     p_db = press_to_db(press[:, ::r_skip_ind])
9339
9340     zind = np.argmin(z - z_plt) # Find the index of the source depth
9351     ax.plot(r_km, p_db[zind, :], label=f"{label}")
9362     ax.set_xlabel("Range (km)")
9373     ax.set_ylabel("Transmission Loss (dB)")
9384     ax.set_ylim(-110, -60)
9395     ax.legend()
9406     ax.grid()
9417     ax.set_title(f"Transmission Loss at {z_plt} m")
9428
9439
9440 def tappert_sqrt(
9451     reference_wavenumber: float,
9462     diff_mat: npt.NDArray,
9473     n_squared: npt.NDArray,
9484 ) -> npt.NDArray:
9495     """Use tappert approximation for sqrt operator.
9506
9517
9528     Approximates sqrt(1+X) where X = 1/k_0^2 * d^2/dz^2 + n^2 - 1
9539     should go to 1 + 0.5 * X for small X
9540     """
9551     x = (1 / reference_wavenumber**2) * diff_mat + diags(n_squared - 1,

```

```

956     0)
957     1:      return np.array(0.5 * x)
958
959
960 def krylov_parabolic_equation(
961     freq: float,
962     z: npt.NDArray[np.float64],
963     z_src: float,
964     r_max: float,
965     dr: float,
966     ssp: npt.NDArray[np.float64],
967     m: int,
968 ) -> tuple[npt.NDArray[np.complex128], npt.NDArray[np.float64]]:
969     """Compute the propagator using Krylov subspace method."""
970     # depth grid
971     wavenumber = 2 * np.pi * freq / ssp
972
973     # range grid
974     r = np.arange(dr, r_max + dr, dr) # Range grid
975
976     # initialize the PE matrix
977     psi = np.zeros((len(z), len(r)), dtype=np.complex128) # PE matrix
978
979     # reference values for PE step
980     ref_ssp = 1500 # Reference sound speed (m/s)
981     ref_wavenumber = 2 * np.pi * freq / ref_ssp # Reference wavenumber
982     (1/m)
983     n_squared = (wavenumber / ref_wavenumber) ** 2
984
985     # starter
986     psi0 = gaussian_starter(ref_wavenumber, z_src, z) # Gaussian
987     starter
988
989     # farfield bessel function
990     d_sq = make_finite_difference_matrix(z) # Finite difference matrix
991
992     q_approx = tappert_sqrt(ref_wavenumber, d_sq, n_squared)
993     exponent = 1j * ref_wavenumber * dr * q_approx
994
995     # ensure not writing as np.matrix
996     if not isinstance(exponent, np.ndarray):
997         exponent = np.asarray(exponent)
998
999     psi[:, 0] = krylov_expm(exponent, psi0, m)
1000    for i in np.arange(0, len(r) - 1, dtype=int):
1001        psi[:, i + 1] = krylov_expm(exponent, psi[:, i], m)
1002
1003    # apply bessel function for range decay
1004    hk = 1 / np.sqrt(r)
1005    press = psi * hk
1006
1007    return press, r.astype(np.float64) # Ensure r is float64 for
1008    consistency
1009
1010
1011 def exact_parabolic_equation(
1012     freq: float,
1013     z: npt.NDArray[np.float64],
1014     z_src: float,
1015     r_max: float,
1016     dr: float,
1017     ssp: npt.NDArray[np.float64],
1018

```

```

10189) -> tuple[npt.NDArray[np.complex128], npt.NDArray[np.float64]]:
10190    """Compute the propagator using Scipy expm as exact method."""
10291    # depth grid
10292    wavenumber = 2 * np.pi * freq / ssp
10293
10294    # range grid
10295    r = np.arange(dr, r_max + dr, dr) # Range grid
10296
10297    # initialize the PE matrix
10298    psi = np.zeros((len(z), len(r)), dtype=np.complex128) # PE matrix
10299
103100   # reference values for PE step
103101   ref(ssp = 1500 # Reference sound speed (m/s)
103102   ref(wavenumber = 2 * np.pi * freq / ref(ssp # Reference wavenumber
103103   (1/m)
103104   n_squared = (wavenumber / ref(wavenumber) ** 2
103105
103106   # starter
103107   psi0 = gaussian_starter(ref(wavenumber, z_src, z) # Gaussian
103108   starter
103109
103110   # farfield bessel function
104111   d_sq = make_finite_difference_matrix(z) # Finite difference matrix
104112
104113   q_approx = tappert_sqrt(ref(wavenumber, d_sq, n_squared)
104114   exponent = 1j * ref(wavenumber * dr * q_approx
104115
104116   # ensure not writing as np.matrix
104117   if not isinstance(exponent, np.ndarray):
104118       exponent = np.asarray(exponent)
104119
104120   psi[:, 0] = expm(exponent) @ psi0
105121   for i in np.arange(0, len(r) - 1, dtype=int):
105122       psi[:, i + 1] = expm(exponent) @ psi[:, i]
105123
105124   # apply bessel function for range decay
105125   hk = 1 / np.sqrt(r)
105126   press = psi * hk
105127
105128   return press, r.astype(np.float64) # Ensure r is float64 for
105129   consistency
105130
```

1059 **Appendix D. Code for Parabolic Equation - Isovelocity.** The following
1060 script was used to simulate the isovelocity profile.

```

10611 from krylov_pe.ocean import (
10622     build_depth_grid,
10633     exact_parabolic_equation,
10644     krylov_parabolic_equation,
10655     plot_pe_result,
10666     plot_pe_tl_at_z,
10677     simple_parabolic_equation,
10688)
10699
107010 import numpy as np
107111 import matplotlib.pyplot as plt
107212
107313
107414 def main():
107515     """Plot the PE results for Munk profile using Crank Nicholson"""
107616     freq = 20 # Frequency (Hz)
107717     z_src = 36 # Source depth (m)
```

```

107&8    clims = (-100, -40)
107&9    dr_exact = 5.0 # Base range in m
108&0    dr_model = 15.0 # Model range step in m
108&1
108&2    z = build_depth_grid(zmax=100, dz=1)
108&3    ssp = 1500 * np.ones_like(z) # Constant sound speed profile (m/s)
108&4
108&5    press_bwd_euler, r_euler = simple_parabolic_equation(
108&6        freq, z, z_src, 3e3, dr_model, ssp, "backward_euler"
108&7    )
108&8    press_crank_nicholson, r_cn = simple_parabolic_equation(
108&9        freq, z, z_src, 3e3, dr_model, ssp, "crank_nicholson"
109&0    )
109&1    press_krylov, r_krylov = krylov_parabolic_equation(
109&2        freq, z, z_src, 3e3, dr_model, ssp, 40
109&3    )
109&4    press_exact, r_exact = exact_parabolic_equation(freq, z, z_src, 3e3,
109&5        dr_exact, ssp)
109&6
109&7    # Plot PE TL contours results
109&8    fig = plt.figure(figsize=(10, 6))
109&9    ax1 = fig.add_subplot(2, 2, 1)
110&0    ax2 = fig.add_subplot(2, 2, 2)
110&1    ax3 = fig.add_subplot(2, 2, 3)
110&2    ax4 = fig.add_subplot(2, 2, 4)
110&3    plot_pe_result(
110&4        ax1,
110&5        r_euler,
110&6        z,
110&7        press_bwd_euler,
110&8        title=r"Backward Euler ($\Delta r = 15$ m)",
110&9        clims=clims,
111&0    )
111&1    plot_pe_result(
111&2        ax2,
111&3        r_cn,
111&4        z,
111&5        press_crank_nicholson,
111&6        title=r"Crank-Nicholson ($\Delta r = 15$ m)",
111&7        clims=clims,
111&8    )
111&9    plot_pe_result(
112&0        ax3,
112&1        r_krylov,
112&2        z,
112&3        press_krylov,
112&4        title=r"Krylov ($\Delta r = 15$ m, $m=40$)",
112&5        clims=clims,
112&6    )
112&7    plot_pe_result(
112&8        ax4,
112&9        r_exact,
113&0        z,
113&1        press_exact,
113&2        title=r"Exact ($\Delta r = 5$ m)",
113&3        clims=clims,
113&4    )
113&5    # ax1.set_xlim(0, 10)
113&6    # ax2.set_xlim(0, 10)
113&7    # ax3.set_xlim(0, 10)
113&8    plt.tight_layout()
113&9

```

```

1140 fig2 = plt.figure(figsize=(10, 6))
1141 ax = fig2.add_subplot(1, 1, 1)
1142 plot_pe_t1_at_z(
1143     ax, r_euler, z, press_bwd_euler, z_src, label="Backward Euler",
1144     dr_plot=5
1145 )
1146 plot_pe_t1_at_z(
1147     ax, r_cn, z, press_crank_nicholson, z_src, label="Crank-
1148 Nicholson", dr_plot=5
1149 )
1150 plot_pe_t1_at_z(ax, r_krylov, z, press_krylov, z_src, label="Krylov"
1151 , dr_plot=10)
1152 plot_pe_t1_at_z(ax, r_exact, z, press_exact, z_src, label="Exact",
1153 dr_plot=5)
1154 ax.set_ylim(clims[0], clims[1])
1155 plt.show()
1156 print("Done!")
1157
1158
1159 if __name__ == "__main__":
1160     main()

```

1161 The following scripts were used for the sensitivity analysis

```

1162 from krylov_pe.ocean import (
1163     build_depth_grid,
1164     exact_parabolic_equation,
1165     krylov_parabolic_equation,
1166     simple_parabolic_equation,
1167 )
1168
1169 import numpy as np
1170 import matplotlib.pyplot as plt
1171
1172 def tl_and_phase(press):
1173     """Calculate the magnitude and phase of the pressure field."""
1174     tl = 20 * np.log10(np.abs(press))
1175     phase = np.angle(press)
1176     return tl, phase
1177
1178
1179 def main():
1180     """Plot the PE results for Munk profile using Crank Nicholson"""
1181     freq = 20 # Frequency (Hz)
1182     z_src = 36 # Source depth (m)
1183     z_plt = z_src # Depth to plot (m)
1184
1185     # m_range for krylov method
1186     m_range = np.arange(10, 101, 10)
1187
1188     # freq = 20 # Frequency (Hz)
1189     # z_src = 1000 # Source depth (m)
1190     # clims = (-110, -60)
1191
1192     dr_model = 30 # Model range in m
1193     ref_wavelength = 1500 / freq # Reference wavelength (m)
1194     dr_exact = 37.5 # Base range step in m
1195     dr_model = 37.5 # Model range step in m
1196     r_max = 3e3 # Maximum range in m
1197
1198     z = build_depth_grid(zmax=100, dz=1)
1199     ssp = 1500 * np.ones_like(z) # Constant sound speed profile (m/s)

```

```

1201<0 # z = build_depth_grid(zmax=5000, dz=10)
1202<1 # ssp = munk_profile(z)
1203<2
1204<3 press_exact, r_exact = exact_parabolic_equation(
1205<4     freq, z, z_src, r_max, dr_exact, ssp
1206<5 )
1207<6 press_bwd_euler, r_euler = simple_parabolic_equation(
1208<7     freq, z, z_src, r_max, 37.5, ssp, "backward_euler"
1209<8 )
1210<9 press_crank_nicholson, r_cn = simple_parabolic_equation(
1211<0     freq, z, z_src, r_max, 37.5, ssp, "crank_nicholson"
1212<1 )
1213<2
1214<3 # plot the phase and magnitude error at z_plt compared to the exact
1215<4 solution
1216<4
1217<5 # get the variables at z_plt
1218<6 zind = np.argmin(z - z_plt)
1219<7 tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
1220<8 tl_bwd_euler, phase_bwd_euler = tl_and_phase(press_bwd_euler[zind,
1221<9 :])
1222<9 tl_crank_nicholson, phase_crank_nicholson = tl_and_phase(
1223<0     press_crank_nicholson[zind, :]
1224<1 )
1225<2
1226<3 # interpolate the exact solution to the model range
1227<4 tl_exact_interp = np.interp(r_euler, r_exact, tl_exact)
1228<5 phase_exact_interp = np.interp(r_euler, r_exact, phase_exact)
1229<6
1230<7 # calculate the RMSE for each method
1231<8 tl_rmse_bwd_euler = np.sqrt(np.mean((tl_exact_interp - tl_bwd_euler)
1232<9 ** 2))
1233<9 tl_rmse_crank_nicholson = np.sqrt(
1234<0     np.mean((tl_exact_interp - tl_crank_nicholson) ** 2)
1235<1 )
1236<2
1237<3 phase_rmse_bwd_euler = np.sqrt(np.mean((phase_exact_interp -
1238<4 phase_bwd_euler) ** 2))
1239<4 phase_rmse_crank_nicholson = np.sqrt(
1240<5     np.mean((phase_exact_interp - phase_crank_nicholson) ** 2)
1241<6 )
1242<7
1243<8 tl_rmse_krylov = np.zeros_like(m_range, dtype=float)
1244<9 phase_rmse_krylov = np.zeros_like(m_range, dtype=float)
1245<0 for i, m in enumerate(m_range):
1246<1     press_krylov, r_krylov = krylov_parabolic_equation(
1247<2         freq,
1248<3         z,
1249<4         z_src,
1250<5         r_max,
1251<6         dr_model,
1252<7         ssp,
1253<8         m=int(m),
1254<9     )
1255<0     tl_exact_interp_kry = np.interp(r_krylov, r_exact, tl_exact)
1256<1     phase_exact_interp_kry = np.interp(r_krylov, r_exact,
1257<2     phase_exact)
1258<2     tl_krylov, phase_krylov = tl_and_phase(press_krylov[zind, :])
1259<3     tl_rmse_krylov[i] = np.sqrt(np.mean((tl_exact_interp_kry -
1260<4     tl_krylov) ** 2))
1261<4     phase_rmse_krylov[i] = np.sqrt(
1262<5     np.mean((phase_exact_interp_kry - phase_krylov) ** 2)

```

```

1263 6      )
1264 7
1265 8 fig = plt.figure(figsize=(10, 6))
1266 9 # plot magnitude RMSE
1267 0 ax1 = fig.add_subplot(1, 1, 1)
1268 1 # ax1 = fig.add_subplot(2, 1, 1)
1269 2 ax1.plot(
1270 3     [m_range[0], m_range[-1]],
1271 4     [tl_rmse_bwd_euler, tl_rmse_bwd_euler],
1272 5     "r--",
1273 6     label="bwd Euler",
1274 7 )
1275 8 ax1.plot(
1276 9     [m_range[0], m_range[-1]],
1277 0     [tl_rmse_crank_nicholson, tl_rmse_crank_nicholson],
1278 1     "b--",
1279 2     label="Crank-Nicholson",
1280 3 )
1281 4 ax1.plot(m_range, tl_rmse_krylov, "ko-", label="Krylov")
1282 5 ax1.set_xlabel("Krylov subspace dimension (m)")
1283 6 ax1.set_ylabel("TL RMSE (dB)")
1284 7 ax1.set_title(
1285 8     f"TL RMSE at z = {z_plt} m ($\Delta r / \lambda_0$ = {dr_model /
1286 9     ref_wavelength:0.3f})"
1287 0 )
1288 0
1289 1 # plot the phase RMSE
1290 2 # ax2 = fig.add_subplot(2, 1, 2)
1291 3 # ax2.plot(
1292 4     [m_range[0], m_range[-1]],
1293 5     [phase_rmse_bwd_euler, phase_rmse_bwd_euler],
1294 6     "r--",
1295 7     label="bwd Euler",
1296 8 )
1297 9 # ax2.plot(
1298 0     [m_range[0], m_range[-1]],
1299 1     [phase_rmse_crank_nicholson, phase_rmse_crank_nicholson],
1300 2     "b--",
1301 3     label="Crank-Nicholson",
1302 4 )
1303 5 # ax2.plot(m_range, phase_rmse_krylov, "ko-", label="Krylov")
1304 6 # ax2.set_xlabel("Krylov subspace dimension (m)")
1305 7 # ax2.set_ylabel("Phase RMSE (radians)")
1306 8 # ax2.set_title("Phase RMSE at z = %.1f m" % z_plt)
1307 9
1308 0 ax1.legend(loc="best")
1309 1 ax1.grid()
1310 2 # ax2.grid()
1311 3 plt.tight_layout()
1312 4 plt.show(block=True)
1313 5
1314 6
1315 7 if __name__ == "__main__":
1316 8     main()

1317 1 from krylov_pe.ocean import (
1318 2     build_depth_grid,
1319 3     exact_parabolic_equation,
1320 4     krylov_parabolic_equation,
1321 5     simple_parabolic_equation,
1322 6 )
1323 7
1324 8 import numpy as np

```

```

1325 9 import matplotlib.pyplot as plt
1326 10
1327 11
1328 12 def tl_and_phase(press):
1329 13     """Calculate the magnitude and phase of the pressure field."""
1330 14     tl = 20 * np.log10(np.abs(press)) # transmission loss in dB
1331 15     phase = np.angle(press)
1332 16     return tl, phase
1333 17
1334 18
1335 19 def main():
1336 20     """Plot the PE results for Munk profile using Crank Nicholson"""
1337 21     freq = 20 # Frequency (Hz)
1338 22     z_src = 36 # Source depth (m)
1339 23     z_plt = z_src # Depth to plot (m)
1340 24
1341 25     m = 70 # Krylov subspace dimension - from the m_sweep script
1342 26     dr_exact = 5.0 # Base range in m
1343 27     r_max = 3e3 # Maximum range in m
1344 28
1345 29     z = build_depth_grid(zmax=100, dz=1)
1346 30     ssp = 1500 * np.ones_like(z) # Constant sound speed profile (m/s)
1347 31
1348 32     # Loop to check the effects of dr on the different methods
1349 33     ref_wavelength = 1500 / freq # Reference wavelength (m)
1350 34     max_wavelength_multiplier = 10
1351 35     dr_range = np.linspace(
1352 36         1, max_wavelength_multiplier * dr_exact, 21, dtype=np.float64
1353 37     )
1354 38
1355 39     # exact solution should not be affected by dr
1356 40     press_exact, r_exact = exact_parabolic_equation(
1357 41         freq, z, z_src, r_max, dr_exact, ssp
1358 42     )
1359 43
1360 44     # initialize RMSE arrays
1361 45     tl_rmse_bwd_euler = np.zeros_like(dr_range, dtype=float)
1362 46     phase_rmse_bwd_euler = np.zeros_like(dr_range, dtype=float)
1363 47     tl_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
1364 48     phase_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
1365 49     tl_rmse_krylov = np.zeros_like(dr_range, dtype=float)
1366 50     phase_rmse_krylov = np.zeros_like(dr_range, dtype=float)
1367 51
1368 52     for i, dr_model in enumerate(dr_range):
1369 53         press_bwd_euler, r_euler = simple_parabolic_equation(
1370 54             freq, z, z_src, r_max, dr_model, ssp, "backward_euler"
1371 55         )
1372 56         press_crank_nicholson, r_cn = simple_parabolic_equation(
1373 57             freq, z, z_src, r_max, dr_model, ssp, "crank_nicholson"
1374 58         )
1375 59         press_krylov, r_krylov = krylov_parabolic_equation(
1376 60             freq,
1377 61             z,
1378 62             z_src,
1379 63             r_max,
1380 64             dr_model,
1381 65             ssp,
1382 66             m=int(m),
1383 67         )
1384 68
1385 69     # plot the phase and magnitude error at z_plt compared to the
1386 70     # exact solution

```

```

13870
13881     # get the variables at z_plt
13892     zind = np.argmin(z - z_plt)
13903     tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
13914     tl_bwd_euler, phase_bwd_euler = tl_and_phase(press_bwd_euler[
1392     zind, :])
13935     tl_crank_nicholson, phase_crank_nicholson = tl_and_phase(
13946         press_crank_nicholson[zind, :]
13957     )
13968
13979     # interpolate the exact solution to the model range
13980     tl_exact_interp = np.interp(r_euler, r_exact, tl_exact)
13991     phase_exact_interp = np.interp(r_euler, r_exact, phase_exact)
14002
14013     # calculate the RMSE for each method
14024     tl_rmse_bwd_euler[i] = np.sqrt(np.mean((tl_exact_interp -
1403     tl_bwd_euler) ** 2))
14045     tl_rmse_crank_nicholson[i] = np.sqrt(
14056         np.mean((tl_exact_interp - tl_crank_nicholson) ** 2)
14067     )
14078
14089     phase_rmse_bwd_euler[i] = np.sqrt(
14090         np.mean((phase_exact_interp - phase_bwd_euler) ** 2)
14101     )
14112     phase_rmse_crank_nicholson[i] = np.sqrt(
14123         np.mean((phase_exact_interp - phase_crank_nicholson) ** 2)
14134     )
14145
14156     tl_krylov, phase_krylov = tl_and_phase(press_krylov[zind, :])
14167     tl_rmse_krylov[i] = np.sqrt(np.mean((tl_exact_interp - tl_krylov
1417 ) ** 2))
14188     phase_rmse_krylov[i] = np.sqrt(
14199         np.mean((phase_exact_interp - phase_krylov) ** 2)
14200     )
14211
14222     fig = plt.figure(figsize=(10, 6))
14233     # plot magnitude RMSE
14244     ax1 = fig.add_subplot(1, 1, 1)
14255     # ax1 = fig.add_subplot(2, 1, 1)
14266     ax1.plot(
14277         dr_range / dr_exact,
14288         tl_rmse_bwd_euler,
14299         "ro-",
14300         label="Backward Euler",
14311     )
14322     ax1.plot(
14333         dr_range / dr_exact,
14344         tl_rmse_crank_nicholson,
14355         "bo-",
14366         label="Crank-Nicholson",
14377     )
14388     ax1.plot(dr_range / dr_exact, tl_rmse_krylov, "ko-", label="Krylov")
14399     ax1.set_xlabel(r"$\Delta r / \Delta r_{exact}$")
14400     ax1.set_ylabel("TL RMSE (dB)")
14411     ax1.set_title("TL RMSE at z = %.1f m" % z_plt)
14422
14433     # plot the phase RMSE
14444     # ax2 = fig.add_subplot(2, 1, 2)
14455     # ax2.plot(
14466         # dr_range / ref_wavelength,
14477         # phase_rmse_bwd_euler,
14488         # "ro-",

```

```

14409      #     label="bwd Euler",
14500      # )
14511      # ax2.plot(
14522      #     dr_range / ref_wavelength,
14533      #     phase_rmse_crank_nicholson,
14544      #     "bo-",
14555      #     label="Crank-Nicholson",
14566      # )
14577      # ax2.plot(dr_range / ref_wavelength, phase_rmse_krylov, "ko-",
14588      label="Krylov")
14599      # ax2.set_xlabel(r"\Delta r / \lambda_0")
14600      # ax2.set_ylabel("Phase RMSE (radians)")
14611      # ax2.set_title("Phase RMSE at z = %.1f m" % z_plt)
14621
14632      ax1.legend(loc="best")
14643      ax1.grid()
14654      # ax2.grid()
14665      plt.rc("text", usetex=True)
14676      plt.tight_layout()
14687      plt.show(block=True)
14698
14709
14710 if __name__ == "__main__":
14721     main()

1473 1 from krylov_pe.ocean import (
1474 2     build_depth_grid,
1475 3     exact_parabolic_equation,
1476 4     krylov_parabolic_equation,
1477 5     simple_parabolic_equation,
1478 6 )
1479 7
1480 8 import numpy as np
1481 9 import matplotlib.pyplot as plt
1482 0
1483 1
1484 2 def tl_and_phase(press):
1485 3     """Calculate the magnitude and phase of the pressure field."""
1486 4     tl = 20 * np.log10(np.abs(press))
1487 5     phase = np.angle(press)
1488 6     return tl, phase
1489 7
1490 8
1491 9 def main():
1492 0     """Plot the PE results for Munk profile using Crank Nicholson"""
1493 1     freq = 100 # Frequency (Hz)
1494 2     z_src = 36 # Source depth (m)
1495 3     z_plt = z_src # Depth to plot (m)
1496 4
1497 5     # m_range for krylov method
1498 6     m_range = np.arange(20, 101, 5)
1499 7
1500 8     # Loop to check the effects of dr on the different methods
1501 9     ref_wavelength = 1500 / freq # Reference wavelength (m)
1502 0     max_wavelength_multiplier = 4
1503 1     dr_range = np.linspace(
1504 2         1, max_wavelength_multiplier * ref_wavelength, 21, dtype=np.
1505 3         float64
1506 4     )
1507 5
1508 6     # freq = 20 # Frequency (Hz)
1509 7     # z_src = 1000 # Source depth (m)
1510 8     # clim = (-110, -60)

```

```

15118
15129 # dr_model = 5 # Model range in m
15130 dr_exact = 5.0 # Base range in m
15141 r_max = 3e3 # Maximum range in m
15152
15163 z = build_depth_grid(zmax=100, dz=1)
15174 ssp = 1500 * np.ones_like(z) # Constant sound speed profile (m/s)
15185 # z = build_depth_grid(zmax=5000, dz=10)
15196 # ssp = munk_profile(z)
15207
15218 press_exact, r_exact = exact_parabolic_equation(
15229     freq, z, z_src, r_max, dr_exact, ssp
1530)
15311
15322 # get the variables at z_plt
15333 zind = np.argmin(z - z_plt)
15344 tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
15355
15366 # initialize RMSE arrays
15377 tl_rmse_fwd_euler = np.zeros_like(dr_range, dtype=float)
15388 phase_rmse_fwd_euler = np.zeros_like(dr_range, dtype=float)
15399 tl_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
15300 phase_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
15311
15322 tl_rmse_krylov = np.zeros((len(m_range), len(dr_range)), dtype=float)
15333
15344 phase_rmse_krylov = np.zeros((len(m_range), len(dr_range)), dtype=
15355 float)
15366 for i, m in enumerate(m_range):
15377     for j, dr_model in enumerate(dr_range):
15388         # plot the phase and magnitude error at z_plt compared to
15399 the exact solution
15400     press_fwd_euler, r_euler = simple_parabolic_equation(
15411         freq, z, z_src, r_max, dr_model, ssp, "backward_euler"
15422     )
15433     press_crank_nicholson, r_cn = simple_parabolic_equation(
15444         freq, z, z_src, r_max, dr_model, ssp, "crank_nicholson"
15455     )
15466     # get the variables at z_plt
15477     zind = np.argmin(z - z_plt)
15488     tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
15499     tl_fwd_euler, phase_fwd_euler = tl_and_phase(press_fwd_euler
15500 [zind, :])
15511     tl_crank_nicholson, phase_crank_nicholson = tl_and_phase(
15522         press_crank_nicholson[zind, :]
15533     )
15544
15555     # interpolate the exact solution to the model range
15566     tl_exact_interp = np.interp(r_euler, r_exact, tl_exact)
15577     phase_exact_interp = np.interp(r_euler, r_exact, phase_exact
15588 )
15599
15600     # calculate the RMSE for each method
15611     tl_rmse_fwd_euler[j] = np.sqrt(
15622         np.mean((tl_exact_interp - tl_fwd_euler) ** 2)
15633     )
15644     tl_rmse_crank_nicholson[j] = np.sqrt(
15655         np.mean((tl_exact_interp - tl_crank_nicholson) ** 2)
15666     )
15677
15688     phase_rmse_fwd_euler[j] = np.sqrt(
15699         np.mean((phase_exact_interp - phase_fwd_euler) ** 2)
15700 )

```

```

15735 )
15746 phase_rmse_crank_nicholson[j] = np.sqrt(
15757     np.mean((phase_exact_interp - phase_crank_nicholson) ** 
1576 2)
15778 )
15789 press_krylov, r_krylov = krylov_parabolic_equation(
15790     freq,
15801     z,
15802     z_src,
15803     r_max,
15804     dr_model,
15805     ssp,
15806     m=int(m),
15807 )
15808 tl_krylov, phase_krylov = tl_and_phase(press_krylov[zind,
15809 :])
15810     tl_rmse_krylov[i, j] = np.sqrt(np.mean((tl_exact_interp -
15811 tl_krylov) ** 2))
15812     phase_rmse_krylov[i, j] = np.sqrt(
15813         np.mean((phase_exact_interp - phase_krylov) ** 2)
15814     )
15815
15914 fig = plt.figure(figsize=(10, 6))
15915 # plot magnitude RMSE
15916 ax1 = fig.add_subplot(1, 1, 1)
15917 cf = ax1.pcolormesh(
15918     dr_range / ref_wavelength,
16009     m_range,
16010     tl_rmse_krylov,
16011     shading="auto",
16012     cmap="jet",
16013 )
16014 plt.colorbar(cf, ax=ax1, label="dB")
16015 ax1.set_xlabel(r"$\Delta r / \lambda_0 (m)$")
16016 ax1.set_ylabel("Krylov subspace dimension (m)")
16017 ax1.set_title(
16018     f"TL RMSE at z = {z_plt:.1f} m "
16019     + r"($\Delta r_{exact} / \lambda_0$)="
16020     + f"${dr_exact / ref_wavelength:.3f}$"
16021 )
16022
16143 # plot the phase RMSE
16154 # ax2 = fig.add_subplot(2, 1, 2)
16165 # ax2.plot(
16176 #     [m_range[0], m_range[-1]],
16187 #     [phase_rmse_fwd_euler, phase_rmse_fwd_euler],
16198 #     "r--",
16209 #     label="Fwd Euler",
16210 # )
16221 # ax2.plot(
16232 #     [m_range[0], m_range[-1]],
16243 #     [phase_rmse_crank_nicholson, phase_rmse_crank_nicholson],
16254 #     "b--",
16265 #     label="Crank-Nicholson",
16276 # )
16287 # ax2.plot(m_range, phase_rmse_krylov, "ko-", label="Krylov")
16298 # ax2.set_xlabel("Krylov subspace dimension (m)")
16309 # ax2.set_ylabel("Phase RMSE (radians)")
16310 # ax2.set_title("Phase RMSE at z = %.1f m" % z_plt)
16321 # ax2.grid()
16332 plt.tight_layout()
16343 plt.show(block=True)

```

```

16354
16365
16376 if __name__ == "__main__":
16387     main()

```

1639 **Appendix E. Code for Parabolic Equation - Munk.** The following script
 1640 was used to determine TL for the Munk profile case.

```

16411 from krylov_pe.ocean import (
16422     build_depth_grid,
16433     exact_parabolic_equation,
16444     krylov_parabolic_equation,
16455     munk_profile,
16466     plot_pe_result,
16477     plot_pe_tl_at_z,
16488     simple_parabolic_equation,
16499 )
16500
16511 import matplotlib.pyplot as plt
16522
16533
16544 def main():
16555     """Plot the PE results for Munk profile using Crank Nicholson"""
16566     freq = 20 # Frequency (Hz)
16577     z_src = 1000 # Source depth (m)
16588     clims = (-110, -60)
16599
16600     z = build_depth_grid(zmax=5000, dz=10)
16611     ssp = munk_profile(z)
16622     # ssp = 1460 + 0.016 * z # Sound speed profile (m/s)
16633
16644     press_bwd_euler, r = simple_parabolic_equation(
16655         freq, z, z_src, 1e5, 10, ssp, "backward_euler"
16666     )
16677     press_crank_nicholson, _ = simple_parabolic_equation(
16688         freq, z, z_src, 1e5, 10, ssp, "crank_nicholson"
16699     )
16700     press_krylov, _ = krylov_parabolic_equation(freq, z, z_src, 1e5, 10,
16711     ssp, 30)
16722     press_exact, _ = exact_parabolic_equation(freq, z, z_src, 1e5, 10,
16733     ssp)
16742
16753     # Plot PE TL contours results
16764     fig = plt.figure(figsize=(10, 6))
16775     ax1 = fig.add_subplot(2, 2, 1)
16786     ax2 = fig.add_subplot(2, 2, 2)
16797     ax3 = fig.add_subplot(2, 2, 3)
16808     ax4 = fig.add_subplot(2, 2, 4)
16819     plot_pe_result(
16820         ax1,
16831         r,
16842         z,
16853         press_bwd_euler,
16864         title="Backward Euler",
16875         clims=clims,
16886     )
16897     plot_pe_result(
16908         ax2,
16919         r,
16920         z,
16931         press_crank_nicholson,
16942         title="Crank-Nicholson",

```

```

16953     clims=clims,
16964 )
16975 plot_pe_result(
16986     ax3,
16997     r,
17008     z,
17019     press_krylov,
17020     title="Krylov",
17031     clims=clims,
17042 )
17053 plot_pe_result(
17064     ax4,
17075     r,
17086     z,
17097     press_exact,
17108     title="Exact",
17119     clims=clims,
17120 )
17131 # ax1.set_xlim(0, 10)
17142 # ax2.set_xlim(0, 10)
17153 # ax3.set_xlim(0, 10)
17164 plt.tight_layout()

17186 fig2 = plt.figure(figsize=(10, 6))
17197 ax = fig2.add_subplot(1, 1, 1)
17208 plot_pe_tl_at_z(
17219     ax, r, z, press_bwd_euler, z_src, label="Backward Euler",
1722 dr_plot=100
17230 )
17241 plot_pe_tl_at_z(
17252     ax, r, z, press_crank_nicholson, z_src, label="Crank-Nicholson",
1726 dr_plot=100
17273 )
17284 plot_pe_tl_at_z(ax, r, z, press_krylov, z_src, label="Krylov",
1729 dr_plot=100)
17305 plot_pe_tl_at_z(ax, r, z, press_exact, z_src, label="Exact", dr_plot
1731 =100)
17326 ax.set_ylim(clims[0], clims[1])
17337 plt.show()
17348 print("Done!")
17359
17360
17371 if __name__ == "__main__":
17382     main()

```