NTNU
Department of Mathematical Sciences

A Krylov projection method for the heat equation

Sindre Eskeland

August 23, 2015

PROJECT

Department of Mathematical Sciences

Norwegian University of Science and Technology

Supervisor: Professor Elena Celledoni

Preface

This is a project thesis as a part of the study program industrial mathematics. It was written during the summer of 2015. It is assumed that the reader is known with numerical difference methods and numerical linear algebra.

Acknowledgment

Thanks to Elena Celledoni for guiding me, and giving me the notes(1) I have shamelessly copied without referencing.

Summary and Conclusions

Solving partial differential equations with finite difference methods often requires performing operations on huge linear systems. The Krylov projection method can reduce the size of the linear system, thereby decreasing memory demand. In this text the Krylov projection method will be tested on the heat equation against other methods in convergence and computation time. The results point to a slight advantage for KPM in some cases, but is in general slower than other methods.

Contents

Chapter 1

Introduction

In this text we will investigate how well the Krylov projection method (KPM) can be used to solve linear differential equations in the form $q' = Aq + f(t)$. These problems arise for example when discretizing time dependent, linear partial differential equations with the method of lines, and have therefore a wide range of applications. KPM is an orthogonal projection technique, the method is explained in detail in section 2. We use Arnoldi's which allows us to choose the size of the orthogonal space, this gives rise to two slightly different methods, presented in detail in section 2.2 and 2.3 respectively.

The main focus of this report is to solve the heat equation with KPM, and compare convergence and computation time with an alternative solution technique, the alternative will be presented in section 3.3. We state the heat equation here with boundary conditions for future references.

$$\begin{aligned} \frac{\partial u(t,x,y)}{\partial t} - \nabla^2 u(t,x,y) &= p(t,x,y) && t \in [0,T] \\ u(0,x,y) &= 0 \\ u &= 0 && \text{on } \partial[0,L] \times [0,L] \end{aligned} \tag{1.1}$$

$p(t,x,y)$ is a smooth function, and $u(t,x,y)$ is the solution we seek.

Let us for now assume that the right hand side of equation (1.1) is separable, so that $p(t,x,y) = f(t)g(x,y)$. Given a vector $v = [v_1, v_2, \cdots, v_m] \in \mathbb{R}^m$ with elements $g(x_i, y_j)$, $x_i, y_j \in [0,L]$ and a

matrix A as an approximation of the Laplacian, we can write the heat equation on the form

$$\begin{aligned} q'(t) - Aq(t) &= f(t)v, \qquad t \in [0,T] \\ q(0) &= 0 \end{aligned} \tag{1.2}$$

where A is an $m \times m$ matrix assumed to be time independent, $f(t)$ is continuous on $[0,T]$, $v \in \mathbb{R}^m$, $q(t)$ is the unknown vector, for $t \in [0,T]$. Note also that $f(t)$ is a scalar function, so that $f(t)v = [f(t)v_1, f(t)v_2, \cdots, f(t)v_m]$. The solution is

$$q(t) = \int_0^t \exp(A(t-s))f(s)vds \tag{1.3}$$

More details about A, x_i, y_i and $q(t)$ will be given in section 3.

Chapter 2

Krylov subspace and methods

We present the Krylov subspace in section 2.1, derive KPM for the heat equation in section 2.2 and 2.3, and show how we can use KPM when p is not separable in section 3.3.

2.1 Krylov subspace

The Krylov subspace is the space consisting of orthonormal vectors $W_n(A, v) = \{v, Av, \cdots, A^{n-1}v\} = \{v_1, v_2, \cdots v_n\}$, where $n \leq m$. The vectors v_i together with $h_{i,j} = v_i^\top A v_j$, are found by using Arnoldi's algorithm, shown in algorithm 1. Letting V_n be the $m \times n$ matrix consisting of column vectors $\{v_1, v_2, \cdots, v_n\}$ and H_n be the $n \times n$ upper Hessenberg matrix containing all elements $(h_{i,j})_{i,j=1,\cdots,n}$, the following hold (4)

$$AV_n = V_n H_n + h_{n+1,n} v_{n+1} e_n^\top \tag{2.1}$$

$$V_n^\top A V_n = H_n \tag{2.2}$$

$$v_i^\top v_j = \delta_{i,j} \tag{2.3}$$

Here, e_n is the nth canonical vector in $\mathbb{R}^n$ and $\delta_{i,j}$ is Kronecker's delta.

Let us define $\nu \leq m$ as the constant where $W_\nu(A, v)$ spans A, meaning we run Arnoldi's algorithm until the stopping criteria is met.

Algorithm 1 (3)Arnoldi's algorithm

Start with A, $v_1 = v/\|v\|_2$, n
for $j = 1,2,\cdots n$ **do**
  Compute $h_{i,j} = \langle Av_j, v_i\rangle$ for $i = 1,2,\cdots j$
  Compute $w_j := Av_j - \Sigma_{i=1}^{j} h_{i,j} v_i$
  $h_{j+1,j} = \|w_j\|_2$
  if $h_{j+1,j} = 0$ **then**
    STOP
  end if
  $v_{j+1} = w_j / h_{j+1,j}$
end for

2.2 Krylov projection method

Let $z(t) = [z_1(t), z_2(t), \cdots, z_v(t)] \in \mathbb{R}^v$ be the vector satisfying $q(t) = V_v z(t)$. We derive KPM by writing this into equation (1.2), that is

$$\begin{aligned} V_v z'(t) - AV_v z(t) &= f(t)v \\ z(0) &= 0 \end{aligned} \tag{2.4}$$

Multiplying by $V_v^\top$ and using equation (2.2) gives

$$\begin{aligned} z'(t) - H_v z(t) &= f(t) V_v^\top v \\ z(0) &= 0 \end{aligned}$$

Using equation (2.3) and $v = \|v\|_2 v_1$, we get

$$\begin{aligned} z'(t) - H_v z(t) &= \|v\|_2 e_1 f(t) \\ z(0) &= 0 \end{aligned} \tag{2.5}$$

And we are done. By solving equation (2.5) for $z(t)$ and approximate $q(t)$ by $q_v(t) = V_v z(t)$ we get the solution. A step by step description of the algorithm is given in algorithm 2. We will denote the method KPM.

Algorithm 2 Krylov projection method

Start with $A, f(t)$ and v.
Compute $[V_\nu, H_\nu] = \texttt{arnoldi}(A, v)$
Solve $z'(t) = H_\nu z + f(t)\|v\|_2 e_1$ for z
$q_\nu(t) \leftarrow V_\nu z(t)$

Let us now consider the residual of equation (1.2) at $q_n(t)$, that is

$$r_n(t) = f(t)v - q_n'(t) + Aq_n(t)$$

Since

$$r_n(t) = f(t)v - V_n z'(t) + AV_n z(t)$$

using equation (2.1) and (2.5) we get

$$r_n(t) = h_{n+1,n} e_n^\top z(t) v_{n+1} \tag{2.6}$$

Since $h_{n+1,n} = 0$ for some $n = \nu \leq m$, this shows the finite termination of the procedure.

2.3 Restarting the Krylov projection method

If $n < \nu$ so that $h_{n+1,n} \neq 0$, we need to restart the procedure described above. Consider first the following equation

$$\begin{aligned} (q - q_n)'(t) - A(q - q_n)(t) &= r_n \\ (q - q_n)(0) &= 0 \end{aligned} \tag{2.7}$$

where r_n is as in equation (2.6). If we can solve this equation for $(q - q_n)$ we can approach the solution q in an iterative manner.

Let us do this by writing $V_\nu z$ in place of q, and $V_n \zeta$ in place of q_n

$$\begin{aligned} (V_\nu z - V_n \zeta)'(t) - A(V_\nu z - V_n \zeta)(t) &= h_{n+1,n} e_n^\top \zeta(t) v_{n+1} \\ (z - \zeta)(0) &= 0 \end{aligned}$$

Multiplying by $V_v^\top$ and using equation (2.2) gives

$$(z-\zeta)'(t) - H_v(z-\zeta)(t) = V_v^\top h_{n+1,n} e_n^\top \zeta(t) v_{n+1}$$
$$(z-\zeta)(0) = 0$$

Let $\xi(t) = (z-\zeta)(t)$, and simplify

$$\begin{aligned} \xi'(t) - H_v\xi(t) &= h_{n+1,n} e_n^\top \zeta(t) \\ \xi(0) &= 0 \end{aligned} \tag{2.8}$$

This is a formula we can solve. Each restart we generate a new Krylov subspace $W_n(A, v_{n+1})$, solve equation (2.8) for $\xi(t)$ and approximate the solution q by $q_n = V_n\xi(t)$. By summing together q_n, we converge towards the solution q_v. Note that the current value of $\zeta(t)$ equals the previous value of $\xi(t)$, and that $h_{n+1,n}$ is from the previous H_n. See algorithm 3 for a step by step description. We will call n a restart variable, and denote the method with KPM(n).

Algorithm 3 Restarting the Krylov projection method

Start with $A, f(t), v$, n and $i = 0$
Compute $[V_n, H_n, h^i_{n+1,n}, v_{n+1}] = \texttt{arnoldi}(A, v)$
Solve $z' = H_n z + f(t)\|v\|_2 e_1$ for z
$q_n \leftarrow Vz$
$\xi_i \leftarrow z$
while convergence criteria not satisfied **do**
  $i \leftarrow i+1$
  Compute $[V_n, H_n, h^i_{n+1,n}, v_{n+1}] = \texttt{arnoldi}(A, v_{n+1}, n)$
  Solve $\xi_i'(t) = H_n\xi_i(t) + h^{i-1}_{n+1,n} e_n^\top \xi_{i-1}(t)$ for ξ_i
  $q_n(t) \leftarrow q_n + V_n\xi_i(t)$
end while

2.4 When p is not seperable

If we let $P(t)$ be a vector consisting of elements $p(t, x_i, y_j)$, so that $P(t) = [P_1(t), P_2(t), \cdots, P_m(t)]$, and write equation (1.2) as

$$\begin{aligned} q_j'(t) - Aq_j(t) &= P_j(t)e_j \\ q_j(0) &= 0 \\ q(t) &= \sum_{i=1}^{m} q_j(t) \end{aligned} \tag{2.9}$$

where e_j is the jth canonical vector in $\mathbb{R}^m$, we can solve the original equation without requiring separability. An important thing to note here is the need for parallel processing power since we need to solve m problems and not just one.

Chapter 3

Implementation

This section will explain the implementation of the methods. We start by discretization in space and time in section 3.1 and 3.2 respectively, and introduce the method we will compare KPM to in section 3.3. In section 3.4 we present what and how we want to measure interesting factor, together with some information about computers and programs used to generate data. In section 3.5 we state some test problems.

3.1 Discretisation in space

We consider the square $[0,1] \times [0,1]$ and divide each spacial direction into $\rho+2$ piece, each piece having a length of $h_s = 1/(\rho+1)$. Let $x_i = h_s \cdot i$ and $y_i = h_s \cdot j$. Since the boundary conditions are known, we will only calculate with ρ numbers, leaving room on each side for the boundary. v and $P(t)$ need to be found in an ordered fashion. We let $v_{i+\rho j} = g(x_i, y_j)$, and $P(t)_{i+\rho j} = p(t, x_i, y_j)$, where $i, j = 1, 2, \cdots \rho+1$. The Laplacian will be approximated by the five point formula given in equation (3.1) as the matrix A. This is a second order approximation.

$$A = \frac{1}{h_s^2}\begin{bmatrix} T & I & & & \\ I & T & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & T & I \\ & & & I & T \end{bmatrix}, T = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & 1 & -4 \end{bmatrix}, I = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix} \tag{3.1}$$

Notice that $m = \rho^2$.

3.2 Discretisation in time

We will consider the time domain $t \in [0,1]$, and divide it in k pieces, giving each piece a length $h_t = 1/(k-1)$, let $t_l = h_t \cdot l$. Trapezoidal rule(2), given in equation (3.2) will be used to integrate in time.

$$\int_a^b f(t)dt \approx \frac{h}{2}\sum_{l=1}^{N}(f(t_{l+1}) + f((t_l)) \tag{3.2}$$

We will only derive the iteration scheme for equation (1.2), but since all the differential equations are similar, it should be easy to make the few changes necessary to solve the other differential equations discussed. To obtain the iteration scheme we write q instead of f, use equation (1.1) and insert the numerical simplifications above.

$$q(t_{l+1}) - q(t_l) = \int_{t_l}^{t_{l+1}} \frac{\partial q}{\partial t}dt \approx \frac{h}{2}(Aq(t_{l+1}) + f(t_{l+1})v + Aq(t_l) + f(t_l)v) \tag{3.3}$$

Solving for $q(t_{l+1})$ gives the Crank-Nicholson scheme for the heat equation

$$q(t_{l+1}) \approx (I - h_t/2A)^{-1}(q(t_l) + \frac{h_t}{2}(Aq(t_l) + f(t_l)v + f(t_{l+1})v)) \tag{3.4}$$

This is a second order method.

3.3 Direct method

We need to compare KPM to a well known and easy to implement method. For this we will solve equation (3.5) straight forward with trapezoidal rule.

$$q'(t) - Aq(t) = P(t) \tag{3.5}$$

We denote it DM for direct method.

3.4 Measurements and computers

We divide implementations in two cases, separable p, and non separable p. Parallel computations were only implemented for non separable p. For both cases three solvers were implemented: KPM, KPM(n), and DM.

The convergence criteria used in algorithm 3 is to stop when the maximum absolute difference in q_n is less than δ. The error is denoted as ϵ and is defined as the largest absolute difference between the correct solution and the estimated solution. The number of iterations needed for convergence in algorithm 3 is denoted by γ. We will in general be looking at computation time and error, and how this scales with δ.

If nothing else is stated assume $\rho = k = 40$, $n = 1$, $\delta = 10^{-15}$. All timed results are averaged over 2 runs, A was implemented as a sparse matrix

All solvers and problems were implemented in MATLAB R2014b. The computer used runs Ubuntu 14.04 LTS with intel i7-4770 CPU, and 16 GB ram.

The parallel implementations were done with MATLAB's commands `parpool` and `parfor`, see (7) and (8) for more information, `nP` denotes the number of processing units used. We will use speedup and parallel efficiency to investigate the gain by using parallel computation. Speedup is defined as

$$S_{\mathtt{nP}} = \frac{\tau_1}{\tau_{\mathtt{nP}}}$$

and parallel efficiency as

$$\eta_{\mathtt{nP}} = \frac{S_{\mathtt{nP}}}{\mathtt{nP}}$$

where $\tau_{\mathtt{nP}}$ is run time with `nP` processors, $S_{\mathtt{nP}}$ is speedup with `nP` processors, and $\eta_{\mathtt{nP}}$ is parallel efficiency for `nP` processors. Speedup measures how much faster a program runs with `nP` pro-

cessors, ideal speedup is when $S_{\mathtt{nP}} = \mathtt{nP}$. Parallel efficiency measures how well each processor is used. Perfect parallel efficiency occurs when $\eta_{\mathtt{nP}} = 1$.

3.5 Test problems

Two test problems are implemented for the separable case. Equation (3.6) is a symmetric problem, and separable for each variable, we will denote it as `P1`.

$$\begin{aligned} u(t,x,y) &= \frac{t}{t+1}x(x-1)y(y-1) \\ f_1(t) &= \frac{1}{t+1^2} & g_1(x,y) &= x(x-1)y(y-1) \\ f_2(t) &= \frac{-t}{t+1} & g_2(x,y) &= 2x(x-1)+2y(y-1) \end{aligned} \tag{3.6}$$

Equation (3.7) is not symmetric, and non separable for x and y, it also has a combination of polynomials and exponential functions, just to make it test a more general case. This problem will be denoted as `P2`.

$$\begin{aligned} u(t,x,y) =& e^{xy}y(y-1)\sin(\pi x)t\cos(t) \\ f_1(t) =& \cos(t) - t\sin(t) & g_1(x,y) =& e^{xy}y(y-1)\sin(\pi x) \\ f_2(t) =& -t\cos(t) \\ g_2(x,y) =& (y-1)y^3e^{xy}\sin(\pi x) \\ &+ e^{xy}(x^2(y-1)y + x(4y-2)+2)\sin(\pi x) \\ &+ 2\pi(y-1)y^2e^{xy}\cos(\pi x) - \pi^2(y-1)ye^{xy}\sin(\pi x) \end{aligned} \tag{3.7}$$

To obtain the solutions, we need to solve for $f_i(t)g_i(x,y)$, $i = 1,2$ and add the solutions together. Clearly parallel computations could be used to solve these, but this will not be done in this text.

`P1` will also be used in the non separable case with $p(t) = f_1(t)g_1(x,y) + f_2(t)g_2(x,y)$. One additional test problem is implemented for non separable p, this is a symmetric problem consisting of both polynomial and exponential functions, it is given in equation (3.8), and will be

denoted as P3.

$$\begin{aligned} u(t,x,y) &= \sin(xyt)(x-1)(y-1) \\ p(t,x,y) &= t^2(x-1)(y-1)y^2\sin(txy) \\ &\quad -2t(y-1)y\cos(txy)+(x-1)x(y-1)y\cos(txy) \\ &\quad -t(x-1)x(2\cos(txy)-tx(y-1)\sin(txy)) \end{aligned} \tag{3.8}$$

Chapter 4

Computational complexity

We will in section 4.1 and 4.2 briefly compare the computational and memory costs of the different solution strategies discussed.

4.1 Computational complexity

Matrix vector multiplication	$\mathcal{O}(m^2)$ (9)
Matrix inversion	$\mathcal{O}(m^3)$ (9)
Arnoldi's algorithm	$\mathcal{O}(n^2 m)$ (5)
Integration	$\mathcal{O}(k)$

Table 4.1: Computational complexity for some operations. Dimension of the matrices is assumed to be $m \times m$ while n is the restart variable and k is the number of steps in time.

All methods need to preform a matrix inversion, and k matrix vector multiplication, if KPM(n) is used the matrix inversion will be cheaper due to smaller dimensions. If KPM or KPM(n) is used we also need to use Arnoldi's algorithm. If p is not separable and we use KPM or KPM(n) we need to perform all operations mentioned m times, instead of one. An overview over the computational cost of these operations is given in table 4.1. A list of asymptotic computational complexity for the methods is given in table 4.2.

From table 4.2 it is clear that KPM is not going to be the best, not only is it terrible when p is non separable, it also need to perform all operations DM needs in addition to Arnoldi, if p is separable. KPM(n) is the only method that can outperform DM, but it is difficult to say without

Method	Separable p	Non separable p
DM	$\mathcal{O}(km^2 + m^3)$	$\mathcal{O}(km^2 + m^3)$
KPM	$\mathcal{O}(km^2 + m^3)$	$\mathcal{O}(km^3 + m^4)$
KPM(n)	$\mathcal{O}((kn^2 + n^2m + n^3)\gamma)$	$\mathcal{O}((kn^2m + n^2m^2 + n^3m)\gamma)$

Table 4.2: Computational complexity for the methods discussed, γ denotes the number of restarts needed to converge.

knowing how γ scales with m and n. Notice that if $n = 1$ or $n = m$ then KPM and KPM(n) have the same computational complexity if we assume $\gamma \propto m^2/n^2$.

4.2 Memory requirement

A	$\sim 10m$
y	$m \times k$
P	$m \times k$
f	k
v	m
V_n	$n \times n$
H_n	$n \times n$
Inverted matrix, with size $n \times n$	$n \times n$

Table 4.3: List over number needed to store large variables.

All methods need to store A, y and an inverted matrix. DM also need to store P. KPM and KPM(n) need to remember f, v, H_n and V_n. In addition KPM and KPM(n) need to remember P if p is non separable. An overview over the memory requirement for the different variables is given in table 4.3. A complete list over memory requirements for the methods is given in table 4.4.

Method	Separable p	Non separable p
DM	$m^2+2mk+10m$	$m^2+2mk+10m$
KPM	$mk+3m^2+11m+k$	$2mk+3m^2+11m+k$
KPM(n)	$mk+2n^2+k+11m+nm$	$2(mk+n^2)+k+11m+nm$

Table 4.4: Memory requirements for the methods discussed. The values are not given asymptotically to make it easier to distinguish between the different methods.

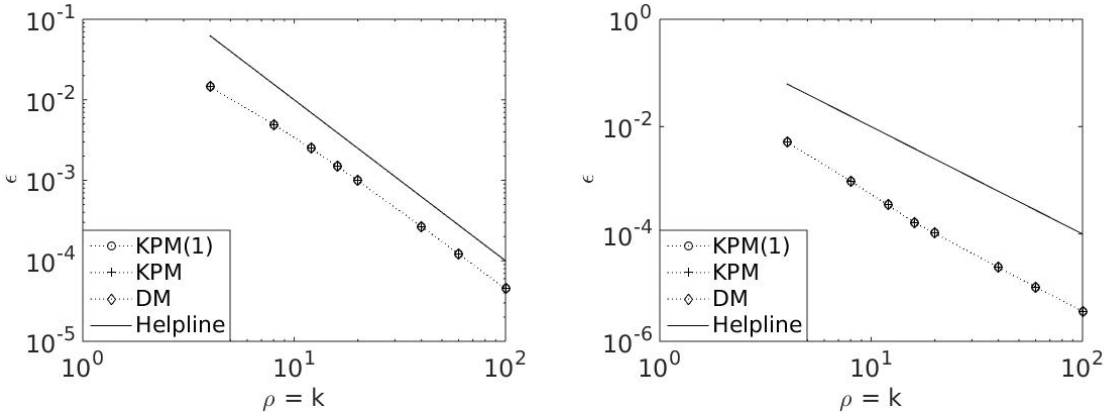
From table 4.4 it is clear that KPM(n) requires the least amount of memory of the three if n is small. Small n is the key, if n is to big we just have KPM, which uses the largest amount of memory of the three. DM is somewhere in the middle, depending on n.

Chapter 5

Results for separable p

We will start by showing correctness of the methods with a convergence plot in section 5.1, then see if we can find any correlation between n and ρ in section 5.3. We compare computation times for the different methods to each other and their predicted computational complexity in section 5.4 and 5.5. We will end by seeing how γ and ϵ scales with δ in section 5.6.

5.1 Convergence

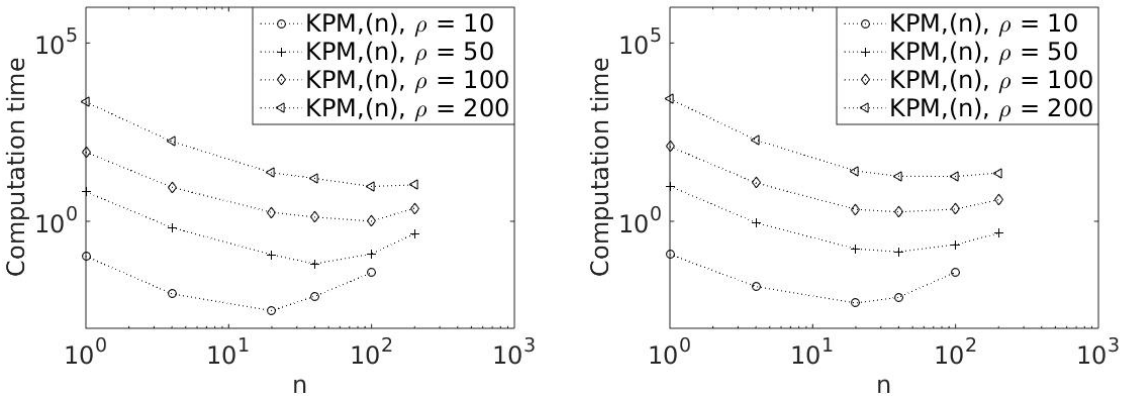


(a) function P1 (b) function P2

Figure 5.1: A convergence plot for several method with $\rho = k$. The helpline shows quadratic convergence.

As can be seen from figure 5.1, all method converges quadratically and overlap perfectly, this shows us that all method preforms as expected regarding convergence.

5.2 Choosing restart variable

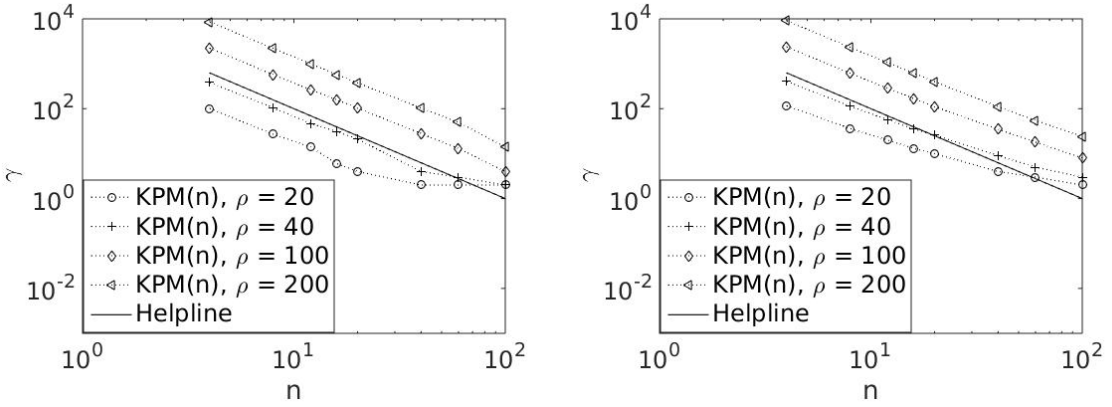


(a) function P1 (b) function P2

Figure 5.2: Computation times plotted against restart variable n.

As we can see from figure 5.2, the optimal restart variable changes as a function of ρ so that larger ρ needs larger n to preform optimally. One point is missing from KPM(n), $\rho = 10$ this is because the last point plotted is the same as KPM. $n = 40$ seams like a good restart variable for these ρ.

5.3 Comparing γ and n

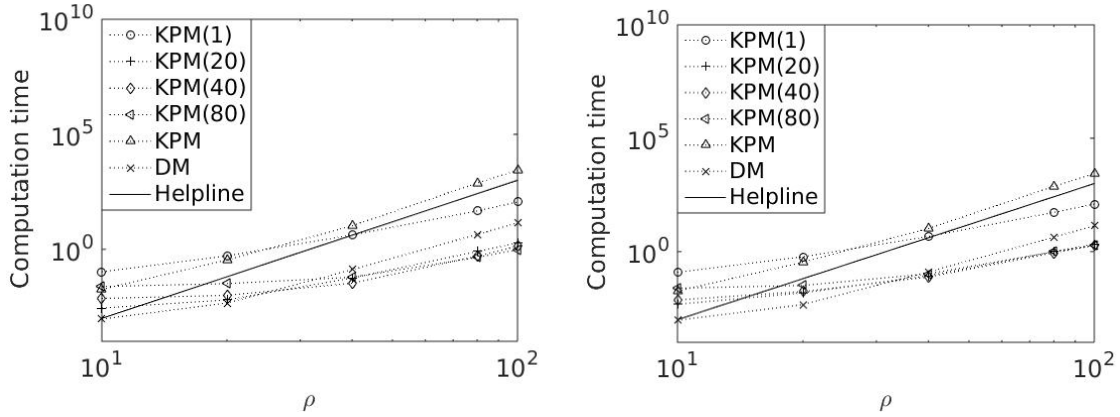


(a) function P1 (b) function P2

Figure 5.3: The number of restarts, γ needed for KPM(n) to converge as a function of n. The helpline decreases with n^2.

As we can see from figure 5.3, γ decreases quadratically when n is small, but decreases less when n is larger. If we put this together with the estimated complexity found in section 5.3, we get that KPM(n) has at most the same estimated complexity as KPM. Notice that the Helpline follows m^2/n^2, as suggested in section 4.1.

5.4 Computation time for m



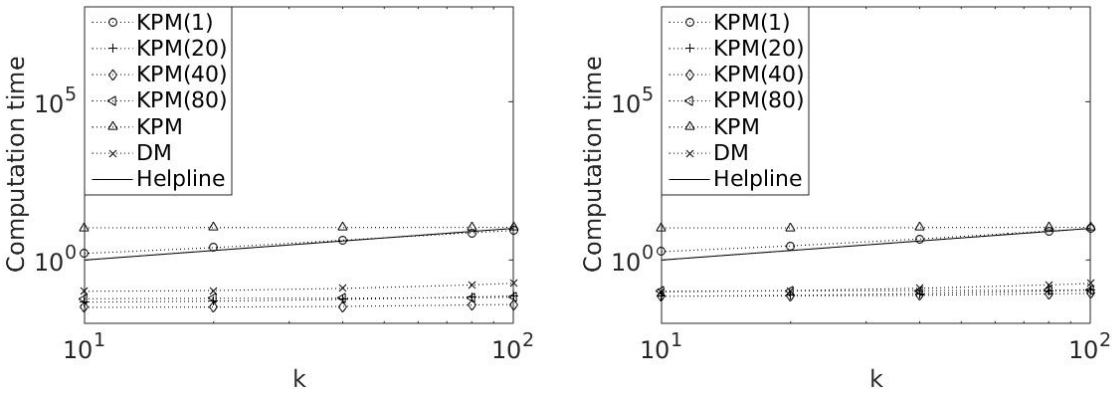
(a) function P1

(b) function P2

Figure 5.4: A plot of computation time as a function of ρ, the helpline increases with $\rho^6 = m^3$.

As we can see from figure 5.4, the computation time for KPM increases as expected, while DM and KPM(n) increases slower, perhaps due to MATLAB's efficient inversion algorithm and less memory demand. Even more interesting is it that KPM(n) is both asymptotically better, and faster than DM for large ρ, so clearly KPM(n) is better in some cases.

5.5 Computation time for k



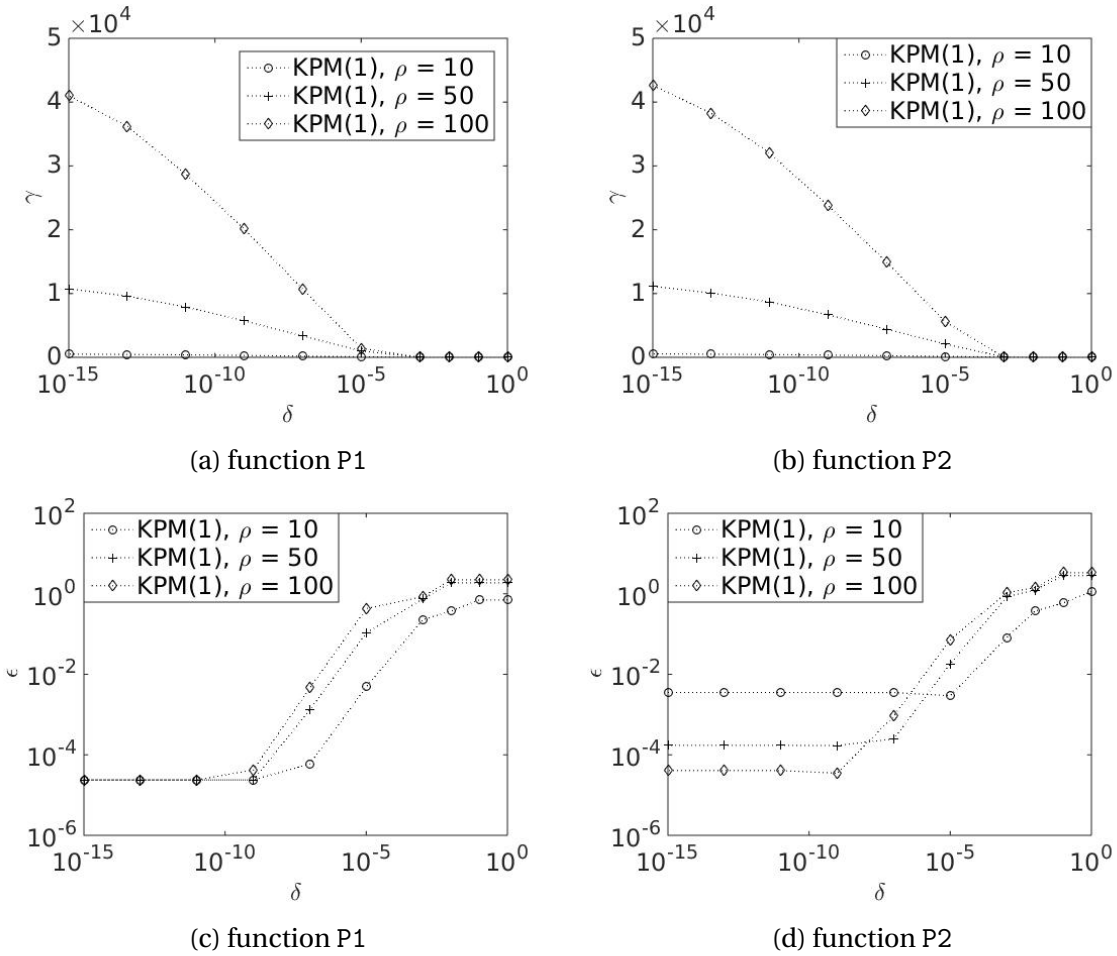
(a) function P1

(b) function P2

Figure 5.5: A plot of computation time as a function of k, the helpline increases with k.

As we can see from figure 5.5, the computation time for almost all tested methods are constant. The reason is that the time needed for initialising is much greater than the time it takes to integrate. The exception is KPM(1), which increases close to linear because relatively more work is done while integrating, due to several restarts. We also see that KPM(n) is faster than DM for some n, KPM is again very slow.

5.6 Comparing δ with γ and ϵ



(a) function P1

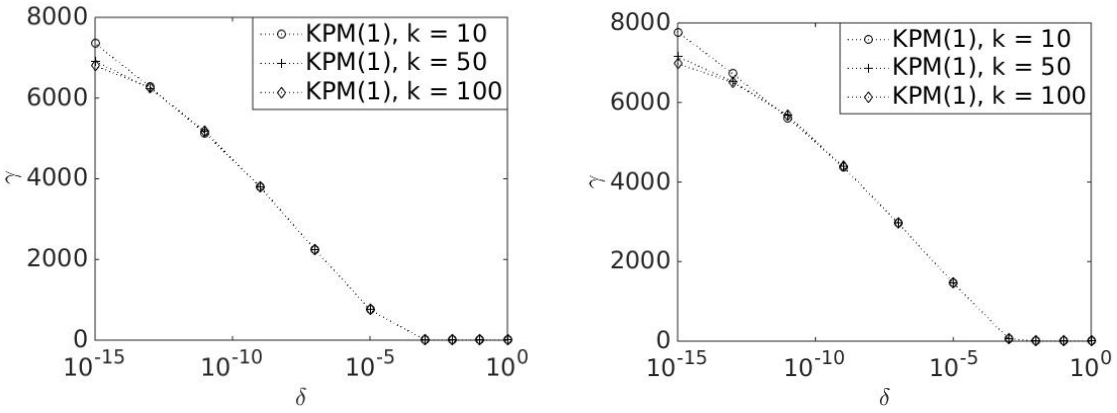
(b) function P2

(c) function P1

(d) function P2

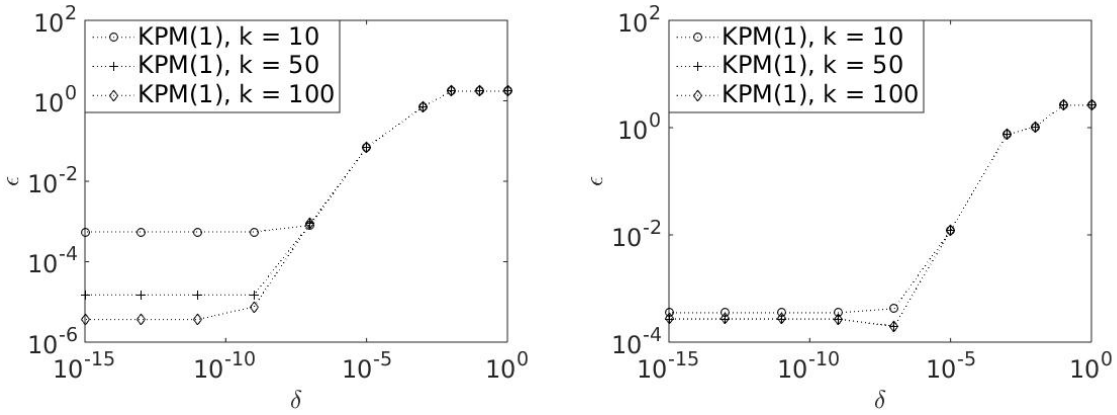
Figure 5.6: A plot of ϵ and γ as a function of δ, with different ρ.

Figure 5.6 and 5.7 shows how γ and ϵ scales with δ. Figure 5.6c has reached to threshold precision with $k = 40$. Both figures shows a loglinear dependence between γ and δ. The figures also shows the importance of choosing an appropriate δ. A lot of time can be saved by choosing δ larger, but precision is lost if δ is to large.



(a) function P1

(b) function P2



(c) function P1

(d) function P2

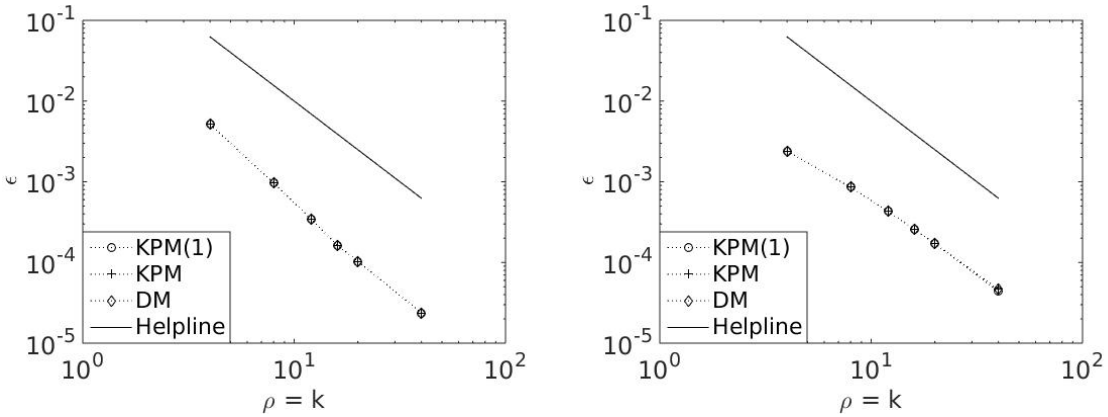
Figure 5.7: A plot of ϵ and γ as a function of δ, with different k.

Chapter 6

Results for non separable p

We start by showing convergence in section 6.1, and proceed with looking into speedup and parallel efficiency in section 6.2. We end by investigating how computation time for the best possible case of KPM compares to DM.

6.1 Convergence



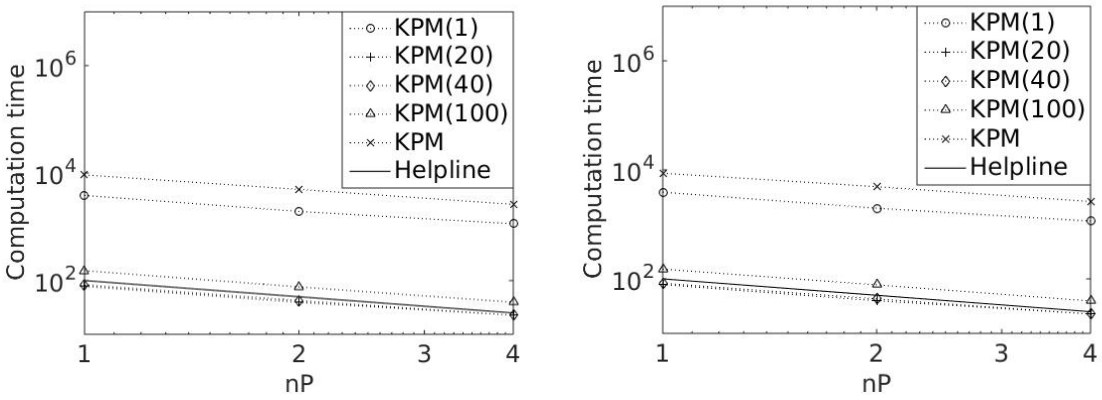
(a) function P1

(b) function P3

Figure 6.1: A convergence plot for several method with $\rho = k$. The helpline shows quadratic convergence.

As can be seen from figure 6.1, all methods converges quadratically and identically, as in the serial case.

6.2 Speedup



(a) function P1 (b) function P3

Figure 6.2: Computation times for several methods with different number of processors. The helpline shows perfect speedup.

	P1		P3	
	nP = 2	nP = 4	nP = 2	nP = 4
KPM	1.8556	3.5234	1.7702	3.3193
KPM(1)	1.9858	3.3740	1.9725	3.3924
KPM(20)	1.9883	3.4525	1.9756	3.4547
KPM(40)	1.9619	3.6667	1.9352	3.6642
KPM(100)	2.0083	3.8618	1.9437	3.8362

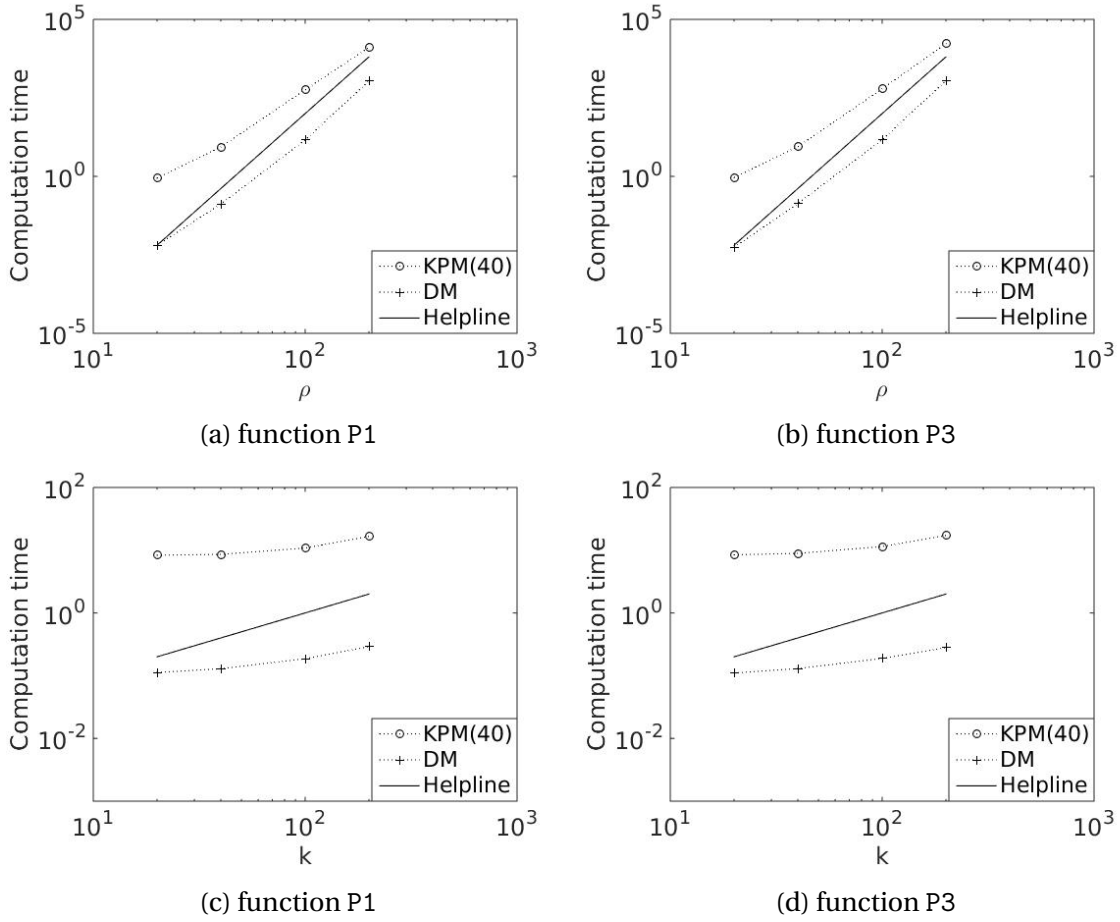
Table 6.1: Speedup for several cases of KPM.

	P1		P3	
	nP = 2	nP = 4	nP = 2	nP = 4
KPM	0.9278	0.8809	0.8851	0.8298
KPM(1)	0.9929	0.8435	0.9862	0.8481
KPM(20)	0.9942	0.8631	0.9878	0.8637
KPM(40)	0.9809	0.9167	0.9676	0.9160
KPM(100)	1.0042	0.9655	0.9719	0.9591

Table 6.2: Parallel efficiency for several cases of KPM.

Figure 6.2 shows good speedup for all methods. Table 6.1 and 6.2 shows that KPM(n) has almost perfect speedup. It is definitely efficient to use several processors on these types of problems. KPM and KPM(1) is not as well suited for this, they probably uses to much memory or too many iterations to be efficient. The optimal value for n with parallel computations seems to be to $n = 100$.

6.3 Comparison



(a) function P1

(b) function P3

(c) function P1

(d) function P3

Figure 6.3: A plot of the computation times tuned to be as efficient as possible for KPM(n). Values are as follows: $n = \rho = k = 40$, $\delta = 10^{-4}$, $\mathtt{nP} = 4$ when nothing else is stated. The helpline for figure 6.3a and 6.3b increases with $\rho^6 = m^3$, while the helpline for figure 6.3c and 6.3d increases with k.

From figure 6.3 it is clear that DM is better in all scenarios simulated here, but KPM(n) is not far behind, with more processing units KPM(40) could have been faster. The other important thing to note here is that computation time for KPM(n) does not increase faster than for DM. It is difficult to say what would happen with larger ρ or k, but they would probably follow the helpline.

One unfortunate simplification done is keeping δ constant while increasing ρ. This might not have a great impact in the result, but as shown in figure 5.6 and 5.7 it is of no use to increase ρ and k without decreasing δ. It would also be of interest to increase n while increasing ρ, as

suggested by figure 5.2.

Chapter 7

Discussion and conclusion

Regarding convergence all methods can perform equally well, the drawback is the need to choose an appropriate δ for KPM and KPM(n). With a larger δ KPM and KPM(n) is less accurate, but with smaller δ we might restart too many times, making the methods inefficient.

The theoretical results shows that the only real candidate to outperform DM is KPM(n), KPM is only near when p is separable. This also holds for memory requirements. In practise it is much the same. KPM did perform overall worst, while KPM(n) and DM was close to each other. KPM(n) is asymptotically better if ρ is large, n is chosen appropriate and p is separable. If p is not separable KPM(n) and DM is asymptotically equal if we keep ρ and n constant, but DM is better. You need several processing units before switching to KPM(n) pays off. Luckily KPM(n) works very well with many processing units. The reason for the high parallel performance is the natural independence in the method. The only communication needed between processors is when adding results, this can be done in $\log_2(\texttt{nP})$ additions. It is also interesting that a good restart variable is also a good value to use with parallel computations.

There is no clear rule to find the best n for each ρ, the only thing that is clear is that larger ρ performs better with larger n. It is worth noting that DM did not work when $\rho > 300$ due to memory shortage, while KPM(n) had no problem with $\rho = 1000$ and $n = 40$, except for excessive computation time.

Further work

To obtain better results KPM should be implemented in a more parallel friendly language, as for example `C`. It would also be a benefit to use a large computer to get data with larger ρ. KPM should also be implemented for other function than the heat equation. There should also be a test where δ and n increases with ρ or k to see how this impacts the results.

My code

If you are interested in any of the code used here you can find it at:

https://github.com/sindreka/Prosjektoppgave

Bibliography

[1] E. Celledoni, I. Moret *A Krylov projection method for system of ODEs* *http://www.sciencedirect.com/science/article/pii/S0168927497000330*

[2] *https://en.wikipedia.org/wiki/Trapezoidal_rule*

[3] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 154, Algorithm 6.1, 2003

[4] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 154, Proposition 6.5, 2003

[5] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 160, 2003

[6] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 171, Proposition 6.10, 2003

[7] *http://se.mathworks.com/help/distcomp/parpool.html*

[8] *http://se.mathworks.com/help/matlab/ref/parfor.html*

[9] *http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations*